Attribute grammars for unranked trees as a query language for structured documents

Frank Neven
Dept. WNI, Infolab, Limburgs Universitair Centrum, Universitaire Campus, B-3590 Diepenbeek, Belgium

Received 30 December 1999; received in revised form 6 October 2004
Available online 9 December 2004

Abstract

Document specification languages, like for instance XML, model documents using extended context-free grammars. These differ from standard context-free grammars in that they allow arbitrary regular expressions on the right-hand side of productions. To query such documents, we introduce a new form of attribute grammars (extended AGs) that work directly over extended context-free grammars rather than over standard context-free grammars. Viewed as a query language, extended AGs are particularly relevant as they can take into account the inherent order of the children of a node in a document. We show that non-circularity remains decidable in EXPTIME and establish the complexity of the non-emptiness and equivalence problem of extended AGs to be complete for EXPTIME. As an application we show that the Region Algebra expressions can be efficiently translated into extended AGs. This translation drastically improves the known upper bound on the complexity of the emptiness and equivalence test for Region Algebra expressions from non-elementary to EXPTIME. Finally, we characterize the expressiveness of extended AGs in terms of monadic second-order logic.

© 2004 Elsevier Inc. All rights reserved.

Keywords: Attribute grammars; Unranked trees; XML; Monadic second-order logic; Expressiveness; Complexity

1. Introduction

Structured document databases can be seen as derivation trees of some grammar which functions as the “schema” of the database [1,2,6,25,27,32,43,50]. Document specification languages like, e.g., XML
[17], model documents using extended context-free grammars. Extended context-free grammars (ECFG) are context-free grammars (CFG) having regular expressions over grammar symbols on the right-hand side of productions. It is known that ECFGs generate the same class of string languages as CFGs. Hence, from a formal language point of view, ECFGs are nothing but shorthands for CFGs. However, when grammars are used to model documents, i.e., when also the derivation trees are taken into consideration, the difference between CFGs and ECFGs becomes apparent: a crucial difference between derivation trees of CFGs and derivation trees of ECFGs is that the former are ranked while the latter are not. In other words, nodes in a derivation tree of an ECFG need not have a fixed maximal number of children. While ranked trees have been studied in depth [22,52], unranked trees only recently received new attention in the context of SGML and XML. Based on work of Pair and Quere [44] and Takahashi [51], Murata defined a bottom–up automaton model for unranked trees [37]. This required describing transition functions for an arbitrary number of children. Murata’s approach is the following: a node is assigned a state by checking the sequence of states assigned to its children for membership in a regular language. In this way, the “infinite” transition function is represented in a finite way. We will extend this idea to attribute grammars. Brügge mann-Klein et al. [13] initiated an extensive study of tree automata over unranked trees. Recently, some of this theory has been applied to XML research [39].

The classical formalism of attribute grammars, introduced by Knuth [35], has always been a prominent framework for expressing computations on derivation trees. Attribute grammars provide a mechanism for annotating the nodes of a tree with so-called “attributes”, by means of so-called “semantic rules” which can work either bottom–up (for so-called “synthesized” attribute values) or top–down (for so-called “inherited” attribute values). This formalism is successfully applied in such diverse fields of computer science as compiler construction and software engineering (for a survey, see [19]). In previous work, we approached attribute grammars from a different direction: we investigated them as a query language for derivation trees of CFGs [42,43].

Inspired by the above-mentioned idea of representing transition functions for automata on unranked trees as regular string languages, we introduce extended attribute grammars (extended AGs) that work directly over ECFGs rather than over standard CFGs. The main difficulty in achieving this is that the right-hand sides of productions contain regular expressions that, in general, specify infinite string languages. This gives rise to two problems for the definition of extended AGs that are not present for standard AGs:

(i) in a production, there may be an unbounded number of grammar symbols for which attributes should be defined; and

(ii) the definition of an attribute should take into account that the number of attributes it depends on may be unbounded.

We resolve these problems in the following way. For (i), we only consider unambiguous regular expressions on the right-hand sides of productions.\(^1\) This means that every child of a node derived by the production \( p = X \rightarrow r \) corresponds to exactly one position in \( r \). We then define attributes uniformly for every position in \( r \) and for the left-hand side of \( p \). For (ii), we only allow a finite set \( D \) as the semantic domain of the attributes and we represent semantic rules as regular languages over \( D \) much in the same way tree automata over unranked trees are defined.

\(^1\)This is no loss of generality, as any regular language can be denoted by an unambiguous regular expression [11]. SGML is even more restrictive as it allows only one-unambiguous regular languages [12,55].
For the definition of inherited attributes, we use regular languages that are included in $D^* # D^*$ where $#$ indicates the position of a node between its siblings. In this way, extended AGs can take into account the inherent order of the children of a node in a document. This makes extended AGs particularly relevant as a query language. Indeed, as argued by Suciu [50], achieving this capability is one of the major challenges when applying the techniques developed for semi-structured data [1] to XML-documents.

An important subclass of queries in the context of structured document databases, are the queries that select those subtrees in a document that satisfy a certain pattern [4,5,33,34,38,45]. These are essentially unary queries: they map a document to a set of its nodes. Extended AGs are especially tailored to express such unary queries: the result of an extended AG consists of those nodes for which the value of a designated attribute equals 1.  

We do make two severe restrictions: (i) we restrict the semantic domain of AGs to be finite sets; and, (ii) we restrict to regular languages to define semantic rules. In our opinion, these restrictions are justified as extended AGs are quite robust: they define precisely the MSO definable unary patterns, a well-studied pattern language [26,41]. Of course, one could try to generalize extended AGs to more powerful semantic rules (e.g., context-free languages) and to infinite domains (the class of all strings). The latter is beyond the scope of this paper.

Related work: Extended AGs as defined in this paper are not the first proposal lifting attribute grammars to extended context-free grammars. Earlier proposals either severely restrict the allowed regular expressions [31], dependencies between attributes [54], or require explicit instantiation, redefinition, finalization, and evaluation rules [10]. The regular right-part attribute grammars of Jüllig and DeRemer [30] are the closest to ours. Essentially, they allow a fixed set of mechanism for specifying attribute dependencies. These include (1) allowing to distribute a string value of a parent either as a list or as a value to its children (or vice versa, from children to parent); (2) passing of attribute values from left to right (or right to left) from one child to another and finally to the parent. The semantics of their approach is obtained through a translation to traditional AGs. Regular right-part attribute grammars are more general as they deal with arbitrary domains, while extended AGs are restricted to finite ones. Nevertheless, extended AGs allow much more flexibility in specifying attribute dependencies within semantics rules.

Contributions: The contributions of this paper can be summarized as follows:

1. We introduce extended attribute grammars as a query language for structured document databases defined by ECFGs. We show that non-circularity, the property that an attribute grammar is well defined for every tree, is in EXPTIME. We obtain an EXPTIME upper bound by reducing the problem to the problem of deciding whether a tree-walking automaton (over unranked trees) cycles. We then show the latter problem to be complete for EXPTIME. The EXPTIME upper bound for the non-circularity test of extended AGs is also a lower bound since deciding non-circularity for standard attribute grammar is already known to be hard for EXPTIME [29].

2. We obtain the exact complexity of some relevant optimization problems for extended AGs. Concretely, we establish the EXPTIME-completeness of the non-emptiness (given an extended AG, does there exist a tree of which a node is selected by this extended AG?) and of the equivalence problem of extended AGs (over the same ECFG). Interestingly, in obtaining this result and the previous complexity result, we make use of non-deterministic two-way automata with a pebble to succinctly describe regular string languages. The crucial property of those, is that they can be transformed into non-deterministic

---

2 We always assume that $D$ contains the values 0 and 1 (false and true).
one-way automata with only exponential size increase, as opposed to the expected double exponential size increase. The latter is a result due to Globerman and Harel [24].

3. We show that Region Algebra expressions (introduced by Consens and Milo [16]) can be simulated by extended AGs. Stated as such, the result is hardly surprising, since the former essentially correspond to a fragment of first-order logic over trees while the latter correspond to full MSO. We, however, exhibit an efficient translation, which gives rise to a drastic improvement on the complexity of the equivalence problem of Region Algebra expressions. To be precise, Consens and Milo first translate each Region Algebra expression into an equivalent first-order logic formula on trees and then invoke the known algorithm testing decidability of such formulas. Unfortunately, the latter algorithm has non-elementary complexity. That is, the complexity of this algorithm cannot be bounded by an elementary function (i.e., an iterated exponential $2^\left(2^{\ldots(2^n)}\right)$ where $n$ is the size of the input). This approach therefore conceals the real complexity of the equivalence test of Region Algebra expressions. Our efficient translation of Region Algebra expressions into extended AGs, however, gives an EXPTIME algorithm when patterns in expressions are represented by DFAs. The thus obtained upper bound more closely matches the already known coNP lower bound [16].

4. We generalize our earlier results on standard attribute grammars [8,43] by showing that extended AGs express precisely the unary queries definable in monadic second-order logic (MSO). The difficult case consists of showing that extended AGs can compute the MSO-equivalence type of each node of the input tree. The only complication, compared to the case of standard attribute grammars, arises from the fact that derivation trees are now unranked.

Result i above is proved in Section i + 4, $i = 1, 2, 3, 4$.

2. Basic definitions

We start by introducing the necessary notions to define extended AGs. More concretely, we recall the definition of unambiguous regular expressions and define tree automata over unranked trees by which extended AGs are inspired.

In all of the following, let $\Sigma$ be a finite alphabet. We denote the length of a string $w$ by $|w|$ and its $i$th letter by $w_i$.

2.1. Unambiguous regular expressions

As is customary, we denote by $L(r)$ the language defined by the regular expression $r$ over $\Sigma$. Further, we denote by $\text{Sym}(r)$ the set of $\Sigma$-symbols occurring in $r$. The marking $\tilde{r}$ of $r$ is obtained by subscripting in $r$ the first occurrence of a symbol of $\text{Sym}(r)$ by 1, the second by 2, and so on. For example, $a_1(a_2 + b_3^*)a_4$ is the marking of $a(a + b^*)a$. We let $|r|$ denote the number of occurrences of $\Sigma$-symbols in $r$, while $r(i)$ denotes the $\Sigma$-symbol at the $i$th occurrence in $r$ for each $i \in \{1, \ldots, |r|\}$. Let $\tilde{\Sigma}$ be the alphabet obtained from $\Sigma$ by subscripting every symbol by all natural numbers, i.e., $\tilde{\Sigma} := \{a_i \mid a \in \Sigma, i \in \mathbb{N}\}$. If $w \in \tilde{\Sigma}^*$ then $w^\#$ denotes the string obtained from $w$ by dropping the subscripts. Observe that $L(\tilde{r})^\# = L(r)$.

In the definition of extended AGs we shall restrict ourselves to unambiguous regular expressions defined as follows:
Definition 2.1. A regular expression $r$ over $\Sigma$ is unambiguous if for all $v, w \in L(\tilde{r})$, $v^\# = w^\#$ implies $v = w$.

That is, $r$ is unambiguous if # is a bijection between $L(\tilde{r})$ and $L(r)$. In other words, every string in $L(r)$ can be matched to $r$ in only one way. For example, the regular expression $a(a + b^*)^a$ is unambiguous while $s := (aa + a)^*$ is not. Indeed, $\tilde{s} = (a_1a_2 + a_3)^*$ and both $a_1a_2, a_3a_3 \in L(\tilde{s})$.

The following proposition, obtained by Book et al. [11], says that the restriction to unambiguous regular expressions is no loss of generality.

Proposition 2.2. For every regular language $R$ there exists an unambiguous regular expression $r$ such that $L(r) = R$.

As usual, a non-deterministic finite automaton $M$ (NFA) over $\Sigma$ is a tuple $(S, \Sigma, \delta, s_0, F)$ where $S$ is finite set of states, $\delta : S \times \Sigma \rightarrow 2^S$ is the transition function, $s_0 \in S$ is the start state, and $F \subseteq S$ is the set of final states. We denote the canonical extension of the transition function to strings by $\delta^*$. A string $w \in \Sigma^*$ is accepted by $M$ if $\delta^*(s_0, w) \cap F \neq \emptyset$. The language accepted by $M$, denoted by $L(M)$, is defined as the set of all strings accepted by $M$. The size of $M$ is defined as $|S| + |\Sigma| + \sum_{s \in S, a \in \Sigma} |\delta(s, a)|$.

A DFA $(S, \Sigma, \delta, s_0, F)$ is an NFA where $|\delta(s, \sigma)| = 1$ for every $s \in S$ and $\sigma \in \Sigma$.

A state assignment $\rho$ of $M$ for a string $w \in \Sigma^*$ is a mapping from $\{1, \ldots, |w|\}$ to $S$. A state assignment $\rho$ for $w$ is valid if $\rho(1) \in \delta(s_0, w_1)$, $\rho(|w|) \in F$, and for $i = 1, \ldots, |w| - 1$, $\rho(i + 1) \in \delta(\rho(i), w_{i+1})$. Clearly, $w$ is accepted by $M$ if and only if there exists a valid state assignment for $w$.

For every unambiguous regular expression $r$ there exists an NFA $M_r$ with the following property: if $w \in L(r)$ then there exists only one valid state assignment of $M_r$ for $w$. That is, $M_r$ can accept $w$ only in one manner. We introduce some more notation to define this automaton $M_r$.

If $w$ is a string and $r$ is an unambiguous regular expression with $w \in L(r)$, then $\tilde{w}_r$ denotes the unique string over $\tilde{\Sigma}$ such that $\tilde{w}_r^\# = w$ and $\tilde{w}_r \in L(\tilde{r})$. For $i = 1, \ldots, |w|$, define $\text{pos}_r(i, w)$ as the subscript of the $i$th symbol in $\tilde{w}_r$. Intuitively, $\text{pos}_r(i, w)$ indicates the position in $r$ matching the $i$th symbol of $w$. For example, if $r = a(b + a)^*$ and $w = abba$, then $\tilde{r} = a_1(b_2 + a_3)^*$ and $\tilde{w}_r = a_1b_2b_2a_3$. Hence,

$$\text{pos}_r(1, w) = 1, \quad \text{pos}_r(2, w) = 2, \quad \text{pos}_r(3, w) = 2 \quad \text{and} \quad \text{pos}_r(4, w) = 3.$$

The following lemma is obtained by Book et al. [11].

Lemma 2.3. For every unambiguous regular expression $r$ there exists an NFA $M_r$ over the states $\{0, \ldots, |r|\}$ with start state 0 such that

1. $L(r) = L(M_r)$;
2. for every string $w \in L(r)$ there exists only one valid state assignment $\rho_w$ of $M_r$ for $w$; and
3. for $i = 1, \ldots, n$, $\rho_w(i) = \text{pos}_r(i, w)$.

Moreover, $M_r$ can be constructed in time polynomial in the size of $r$.

Actually, the fact that $M_r$ can be constructed in time polynomial in the size of $r$ is not stated in [11], but easily follows from an inspection of the proof of Theorem 4 of [11].
Proviso 2.4. In the remaining, when we say regular expression, we always mean unambiguous regular expression.

2.2. Trees

In this paper, we only consider rooted trees where edges are directed from the root to the leaves. Additionally, the children of each node are ordered and each node carries a label from some finite alphabet $\Sigma$. We refer to such trees as $\Sigma$-trees. We introduce some terminology.

Trees will be denoted by the boldface characters $t$, $s$, $s_1$, $\ldots$, while nodes of trees are denoted by $n$, $m$, $n_1$, $\ldots$. We use the following convention: if $n$ is a node of a tree $t$, then $n_i$ denotes the $i$th child of $n$. We denote the set of nodes of $t$ by $\text{Nodes}(t)$ and the root of $t$ by $\text{root}(t)$. Further, the arity of a node $n$ in a tree, denoted by $\text{arity}(n)$, is the number of children of $n$. We say that a tree $t$ has arity $m$, for $m \in \mathbb{N}$, if $\text{arity}(n) \leq m$ for every $n \in \text{Nodes}(t)$. Sometimes we use rank instead of arity, and say that a tree is of rank $m$. The subtree of $t$ rooted at $n$ is denoted by $t_n$; the envelope of $t$ at $n$, that is, the tree obtained from $t$ by deleting the subtrees rooted at the children of $n$ is denoted by $\overline{t_n}$. We denote the label of $n$ in $t$ by $\text{lab}_t(n)$. We denote the empty string by $\varepsilon$.

We end by introducing the following notation. When $\sigma$ is a symbol in $\Sigma$ and $t_1$, $\ldots$, $t_n$ are $\Sigma$-trees, then $\sigma(t_1, \ldots, t_n)$ is the $\Sigma$-tree graphically represented by

$$
\sigma \\
\downarrow \\
\begin{array}{c}
t_1 \\
\cdots \\
\downarrow \\
t_n.
\end{array}
$$

So, the tree consisting of just one node labeled with $\sigma$ is denoted by $\sigma()$. However, we sometimes just denote it with $\sigma$.

Note that in the above definitions there is no a priori bound on the number of children that a node may have. We refer to them as unranked trees.

2.3. Extended context-free grammars

Extended AGs are defined over extended context-free grammars which are defined as follows [3]:

Definition 2.5. An extended context-free grammar (ECFG) is a tuple $G = (N, T, P, U)$, where

- $T$ and $N$ are disjoint finite non-empty sets, called the set of terminals and non-terminals, respectively;
- $U \in N$ is the start symbol; and
- $P$ is a finite set of productions of the form $X \rightarrow r$ where $X \in N$ and $r$ is a regular expression over $N \cup T$. For every $X \in N$, there is at most one production with left-hand side $X$ in $P$.

A derivation tree $t$ over an ECFG $G$ is a tree labeled with symbols from $N \cup T$ such that

- the root of $t$ is labeled with $U$;

---

3 Note that $t_n$ and $\overline{t_n}$ have $n$ in common.
• for every node \( n \) with children \( n_1, \ldots, n_m, m \geq 0 \), there exists a production \( X \rightarrow r \) such that \( n \) is labeled with \( X \), for \( i = 1, \ldots, m \), \( n_i \) is labeled with \( X_i \), and \( X_1 \cdots X_m \in L(r) \); we say that \( n \) is derived by \( X \rightarrow r \).

A leaf node may be labeled with a non-terminal \( X \); in that case, there must be a production \( X \rightarrow r \) with \( \varepsilon \in L(r) \). We denote by \( L(G) \) the set of derivation trees over \( G \). Note that derivation trees of ECFGs are unranked. Throughout this chapter we make the harmless technical assumption that the start symbol does not occur on the right-hand side of a production. The only place we make use of this convention is in the proof of Theorem 8.4.

2.4. Tree automata over unranked trees

We continue with the definition of non-deterministic bottom–up tree automata over unranked trees [13] by which the mechanism of extended AGs is inspired. Interestingly, these automata will also be used to obtain the exact complexity of testing non-emptiness and equivalence of extended AGs in Section 6.

**Definition 2.6.** A non-deterministic bottom–up tree automaton (NBTA) is a tuple \( B = (Q, \Sigma, F, \delta) \), where \( Q \) is a finite set of states, \( F \subseteq Q \) is the set of final states, and \( \delta \) is a function \( Q \times \Sigma \rightarrow 2^Q \) such that \( \delta(q, \sigma) \) is a regular string language over \( Q \) for every \( \sigma \in \Sigma \) and \( q \in Q \). The semantics of \( B \) on a \( \Sigma \)-tree \( t \), denoted by \( \delta^*(t) \subseteq Q \), is defined inductively as follows: if \( t \) consists of only one node labeled with \( \sigma \) then \( \delta^*(\sigma) = \{ q \mid \varepsilon \in \delta(q, \sigma) \} \); if \( t = (t_1, \ldots, t_n) \) then

\[
\delta^*(t) = \{ q \mid \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n) \text{ such that } q_1 \cdots q_n \in \delta(q, \sigma) \}.
\]

A \( \Sigma \)-tree \( t \) is accepted by the automaton \( B \) if \( \delta^*(t) \cap F \neq \emptyset \). The tree language defined by \( B \), denoted by \( L(B) \), consists of the trees accepted by \( B \). A tree language \( T \) is recognizable if there exists an NBTA \( B \) such that \( T = L(B) \).

Further, we say that \( B \) is deterministic when \( \delta(q, \sigma) \cap \delta(q', \sigma) = \emptyset \) for every \( \sigma \in \Sigma \) and \( q, q' \in Q \) with \( q \neq q' \). We use the abbreviation DBTA to refer to such automata.

We represent the string languages \( \delta(q, \sigma) \) by NFAs. The size of \( B \) then is the sum of the sizes of \( Q, \Sigma, \) and the NFAs defining the transition function.

We will use the following notion in Section 6. A state assignment of \( B \) for a tree \( t \) is a mapping \( \rho \) from the nodes of \( t \) to \( Q \). A state assignment is valid if for every node \( n \) of \( t \) of arity \( n \), \( \rho(n1) \cdots \rho(nn) \in \delta(\rho(n), X) \), where \( n \) is labeled with \( X \), and \( \rho(\text{root}(t)) \in F \). Clearly, a tree \( t \) is accepted by \( B \) if and only if there exists a valid state assignment for \( t \).

A detailed study of tree automata over unranked trees has been initiated by Brüggemann-Klein, Murata and Wood [13,37]. Among many things, they show that DBTAs are as expressive as NBTAs and that the recognizable languages are closed under the Boolean operations. The latter also follows by the equivalence to monadic second-order logic [41].

Tree automata are defined over an arbitrary alphabet, but we consider derivation trees of ECFGs in this chapter. This seeming distinction can be dispensed with since we can always restrict an NBTA to the derivation trees of an ECFG as illustrated next. We point out that this lemma is well known for the ranked case with respect to CFGs [22].
Lemma 2.7. Let $G = (N, T, P, U)$ be an ECFG and let $B$ be an NBTA over $\Sigma \subseteq N \cup T$. Then there exists an NBTA $B^G$ such that $L(B^G) = L(G) \cap L(B)$. Moreover, $B^G$ can be constructed in polynomial time.

Proof. We define an NBTA $M$ such that $L(M) = L(G)$. Since recognizable tree languages are closed under the Boolean operations, we can then define $B^G$ as an automaton accepting $L(M) \cap L(B)$. As the intersection of two NBTA can be computed in polynomial time, the time bound follows.

Define $M = (Q, T \cup N, F, \delta)$, where $Q = T \cup N$, $F = \{U\}$, and $\delta$ is defined as follows: for every $\sigma_1 \in T$ and $\sigma_2 \in T \cup N$,

$$\delta(\sigma_1, \sigma_2) := \begin{cases} \{\varepsilon\} & \text{if } \sigma_1 = \sigma_2, \\ \emptyset & \text{otherwise} \end{cases}$$

and for every $X \in N$ and $Y \in N \cup T$:

$$\delta(X, Y) := \begin{cases} L(r) & \text{if } X = Y \text{ and } X \rightarrow r \text{ is in } P, \\ \emptyset & \text{otherwise} \end{cases} \quad \square$$

The proof of the following lemma is a straightforward generalization of the ranked case (see, e.g., the survey paper by Vardi [53]).

Lemma 2.8. Deciding whether the tree language accepted by an NBTA is non-empty is in PTIME.

Proof. Let $B = (Q, \Sigma, F, \delta)$ be an NBTA. We inductively compute the set of reachable states $R$ defined as follows: $q \in R$ iff there exists a tree $t$ with $q \in \delta^*(t)$. Obviously, $L(B) \neq \emptyset$ if and only if $R \cap F \neq \emptyset$. Define for all $n > 0$,

$$R_1 := \{q \in Q \mid \exists \sigma \in \Sigma : \varepsilon \in \delta(q, \sigma)\},$$

$$R_{n+1} := \{q \in Q \mid \exists \sigma \in \Sigma : \delta(q, \sigma) \cap R_n^* \neq \emptyset\}.$$

Note that for all $n$, $R_n \subseteq R_{n+1} \subseteq Q$. Hence, $R_{|Q|} = R_{|Q|+1}$. Thus, define $R$ as $R_{|Q|}$.

Clearly, $R_1$ can be computed in time linear in the size of $B$. Since testing non-emptiness of $\delta(q, \sigma) \cap R_n^*$ can be done in time polynomial in the sum of the sizes of these (see, e.g., [28]), each $R_{n+1}$ can be computed in time polynomial in the size of $B$.

2.5. Two-way automata with a pebble

We conclude by introducing the following important device. A two-way non-deterministic finite automaton with one pebble is an NFA that can move in two directions over the input string and that has one pebble which it can lay down on the input string and pick back up later on. We refrain from giving a formal definition of such automata as we will only use them informally to describe our algorithmic computation. Blum and Hewitt [9] showed that such automata can only define regular languages. In the sequel, we will need the following stronger result obtained by Globerman and Harel [24, Proposition 3.2].
Proposition 2.9. Every two-way non-deterministic finite automaton $M$ with one pebble is equivalent to an NFA $M'$ whose size is exponential in the size of $M$.

By an inspection of the proof it follows that, in fact, the size of $M'$ can be uniformly bounded by a function $|\Sigma| \cdot 2^{q(|S|)}$, where $q$ is a polynomial, $\Sigma$ is the alphabet, and $S$ is the set of states of $M$. Additionally, $M'$ can be constructed in time polynomial in its size.

3. Example

We give a small example introducing the important ideas for the definition of extended attribute grammars in the next section.

First, we briefly illustrate the mechanism of attribute grammars by giving an example of a Boolean-valued standard attribute grammar (BAG). The latter are studied by Neven and Van den Bussche [42,43]. As mentioned in the introduction, attribute grammars provide a mechanism for annotating the nodes of a tree with so-called “attributes”, by means of so-called “semantic rules”. A BAG assigns Boolean values by means of propositional logic formulas to attributes of nodes of input trees. Consider the CFG consisting of the productions $U \rightarrow AA$, $A \rightarrow a$, and $A \rightarrow b$. The following BAG selects the first $A$ whenever the first $A$ is expanded to an $a$ and the second $A$ is expanded to a $b$:

$$U \rightarrow AA \quad \text{select}(1):= \text{is}_a(1) \land \neg \text{is}_a(2),$$
$$A \rightarrow a \quad \text{is}_a(0):= \text{true},$$
$$A \rightarrow b \quad \text{is}_a(0):= \text{false}.$$  

Here, the 1 in select(1) indicates that the attribute select of the first $A$ is being defined. Moreover, this attribute is true whenever the first $A$ is expanded to an $a$ (that is, is_a(1) should be true) and the second $A$ is expanded to a $b$ (that is, is_a(2) should be false). The other rules then define the attribute is_a in the obvious way. In the above, 0 refers to the left-hand side of the production.

Consider the ECFG consisting of the productions $U \rightarrow (A + B)^*$, $A \rightarrow \varepsilon$, and $B \rightarrow \varepsilon$. Suppose, we want to construct an attribute grammar selecting those $A$’s that are preceded by an even number of $A$’s and succeeded by an odd number of $B$’s. Like above we will use rules defining the attribute select. This gives rise to two problems not present for BAGs: (i) $U$ can have an unbounded number of children labeled with $A$ which implies that an unbounded number of attributes should be defined; (ii) the definition of an attribute of an $A$ depends on its siblings, whose number is again unbounded.

We resolve this in the following way. For (i), we just define select uniformly for each node that corresponds to the first position in the regular expression $(A + B)^*$. For (ii), we use regular languages as semantic rules rather than propositional formulas. The following extended AG expresses the above query:

$$U \rightarrow (A + B)^* \quad \text{select}(1):= (\pi_0 = \varepsilon, \pi_1 = \text{lab}, \pi_2 = \text{lab}, \quad R_{\text{true}} = (B^*AB^*AB^*)^*#A^*BA^*(A^*BA^*BA^*), \quad R_{\text{false}} = (A + B + #)^* - R_{\text{true}}).$$

The 1 in select(1) indicates that the attribute select is defined uniformly for every node corresponding to the first position in $(A + B)^*$. In the first part of the semantic rule, each $\pi_i$ lists the attributes of position $i$ that will be used. Here, both for position 1 and 2 this is only the attribute lab which is a special attribute containing the label of the node. As $\pi_0 = \varepsilon$, no attributes for position 0 are used. Consider the input tree
U(AAABB). Then, to check, for instance, whether the third A is selected we enumerate the attributes mentioned in the first part of the rule and insert the symbol # before the node under consideration. This gives us the string

\begin{align*}
1 & 1 1 2 2 2 \text{ position in } (A + B)^* \\
A & A #A B B B \\
1 & 2 3 4 5 6 \text{ position in } AAABB
\end{align*}

The attribute select of the third child will be assigned the value true since the above string belongs to \( R_{\text{true}} \). Note that \((B^*AB^*AB^*)^*\) and \(A^*BA^*(A^*BA^*BA^*)^*\) define the set of strings with an even number of A’s and with an odd number of B’s, respectively. The above will be defined formally in the next section.

4. Attribute grammars over extended context-free grammars

We next define extended attribute grammars (extended AGs) over ECFGs whose attributes can take only values from a finite set \( D \).

**Proviso 4.1.** Unless explicitly stated otherwise, we always assume an ECFG \( G = (N, T, P, U) \). When we say tree we always mean derivation tree of \( G \).

**Definition 4.2.** An attribute grammar vocabulary is a tuple \((D, A, \text{Syn}, \text{Inh})\), where

- \( D \) is a finite set of values called the semantic domain. We assume that \( D \) always contains the Boolean values 0 and 1, and that \( N \cup T \subseteq D \);
- \( A \) is a finite set of symbols called attributes; we always assume that \( A \) contains the attribute lab;
- \( \text{Syn} \) and \( \text{Inh} \) are mappings from \( N \cup T \) to the powerset of \( A \setminus \{\text{lab}\} \) such that for every \( X \in N \)
  \[ \text{Syn}(X) \cap \text{Inh}(X) = \emptyset; \]
  \[ \text{for every } X \in T, \text{Syn}(X) = \emptyset; \] and \( \text{Inh}(U) = \emptyset \).

If \( a \in \text{Syn}(X) \), we say that \( a \) is a synthesized attribute of \( X \). If \( a \in \text{Inh}(X) \), we say that \( a \) is an inherited attribute of \( X \). We also agree that lab is an attribute of every \( X \) (this is a predefined attribute; for each node its value will be the label of that node). The above conditions express that an attribute cannot be a synthesized and an inherited attribute of the same grammar symbol, that terminal symbols do not have synthesized attributes, and that the start symbol does not have inherited attributes.

We now formally define the semantic rules of extended AGs. For a production \( p = X \rightarrow r \), define \( p(0) = X \), and for \( i \in \{1, \ldots, |r|\} \), define \( p(i) = r(i) \). We fix some attribute grammar vocabulary \((D, A, \text{Syn}, \text{Inh})\) in the following definitions.

**Definition 4.3.** 1. Let \( p = X \rightarrow r \) be a production of \( G \) and let \( a \) be an attribute of \( p(i) \) for some \( i \in \{0, \ldots, |r|\} \). The triple \((p, a, i)\) is called a context if \( a \neq \text{lab} \), \( a \in \text{Syn}(p(i)) \) implies \( i = 0 \), and \( a \in \text{Inh}(p(i)) \) implies \( i > 0 \).

2. A rule in the context \((p, a, i)\) is an expression of the form

\[ a(i) := \langle \pi_0, \ldots, \pi_{|r|}; (R_d)_{d \in D} \rangle. \]
where

- for \( j = \{0, \ldots, |r|\} \), \( \pi_j \) is a sequence of attributes of \( p(j) \);
- if \( i = 0 \), then, for each \( d \in D \), \( R_d \) is a regular language over the alphabet \( D \); and
- if \( i > 0 \), then, for each \( d \in D \), \( R_d \) is a regular language over the alphabet \( D \cup \{\#\} \).

For all \( d, d' \in D \), if \( d \neq d' \) then \( R_d \cap R_{d'} = \emptyset \). Further, if \( i = 0 \) then \( \bigcup_{d \in D} R_d = D^* \). If \( i > 0 \) then \( D^*\#D^* \subseteq \bigcup_{d \in D} R_d \).

Note that a \( R_d \) is allowed to contain strings with several or no occurrences of \#. Such strings are irrelevant for the semantics of the extended AGs. We always assume that \( \# \notin D \).

An extended AG is then defined as follows:

**Definition 4.4.** An extended attribute grammar (extended AG) \( \mathcal{F} \) consists of an attribute grammar vocabulary, together with a mapping assigning to each context a rule in that context.

The size of an extended AG is the sum of the sizes of the attribute grammar vocabulary, the ECFG and the size of the semantic rules where we represent the regular languages \( R_d \) by NFAs.

In examples, it will always be understood which rule is associated to which context. We illustrate the above definitions with an example.

**Example 4.5.** Let \( G = (N, T, P, DB) \) be the ECFG with

\[
N = \{DB, Poem, Verse, Word\},
\]

\( T = \{a, \ldots, z\} \), and \( P \) contains the productions

\[
DB \to Poem^* \\
Poem \to Verse^* \\
Verse \to Word^* \\
Word \to (a + \ldots + z)^*.
\]

In Fig. 1 an example of an extended AG \( \mathcal{F} \) over \( G \) is depicted. Recall that every grammar symbol has the attribute lab; for each node this attribute has the label of that node as value. Further, \( A = \{\text{lab, lord, king, king\_lord}\} \). We have \( \text{Syn(Word)} = \{\text{king, lord} \} \), \( \text{Syn(Verse)} = \{\text{king\_lord} \} \), \( \text{Syn(Poem)} = \{\text{result} \} \), \( \text{Inh(Poem)} = \{\text{first} \} \), \( \text{Inh(Verse)} = \text{Inh(Word)} = \emptyset \). The grammar symbols DB, \( a, \ldots, z \) have no attributes apart from lab.

The semantics of this extended AG will be explained below. Here,

\[
D = \{0, 1, a, \ldots, z, DB, Poem, Verse, Word\}.
\]
We use regular expressions to define the languages $R_1$; for the first rule, $R_0$ is defined as $(D \cup \{\#\})^* - R_1$; for all other rules, $R_0$ is defined as $D^* - R_1$; those $R_d$ that are not specified are empty; $\varepsilon$ stands for the empty sequence of attributes.

The semantics of an extended AG is that it defines attributes of the nodes of derivation trees of the underlying grammar $G$. This is formalized next.

**Definition 4.6.** If $t$ is a derivation tree of $G$ then a valuation $v$ of $t$ is a function that maps each pair $(n, a)$, where $n$ is a node in $t$ and $a$ is an attribute of the label of $n$, to an element of $D$, and that maps for every $n$, $(n, \text{lab})$ to the label of $n$.

In the sequel, for a pair $(n, a)$ as above we will use the more intuitive notation $a(n)$. To define the semantics of $F$ we first need the following definition. If $a = a_1 \cdots a_k$ is a sequence of attributes and $n$ is a node of $t$, then define $a(n)$ as the sequence of attribute–node pairs $a(n) = a_1(n) \cdots a_k(n)$.

**Definition 4.7.** Let $t$ be a derivation tree, $n$ a node of $t$, and $a$ an attribute of the label of $n$.

Synthesized: Let $n$ be a node of arity $m$ derived by $p = X \rightarrow r$, and let $(\pi_0, \ldots, \pi_{|\cdot|}; (R_d)_{d \in D})$ be the rule associated to the context $(p, a, 0)$. Define for $l \in \{1, \ldots, m\}$, $j_l = \text{pos}_r(l, w)$, where $w$ is the string formed by the labels of the children of $n_0$. Then define $W(a(n))$ as the sequence

$$\pi_0(n) \cdot \pi_{j_1}(n)_1 \cdot \pi_{j_m}(n)_m.$$  

For each $d$, we denote the language $R_d$ associated to $a(n)$ by $R_d^{a(n)}$.

Inherited: Let $n_1, \ldots, n_{k-1}$ be the left siblings, $n_{k+1}, \ldots, n_m$ be the right siblings, and $n_0$ be the parent of $n$. Let $n_0$ be derived by $p = X \rightarrow r$, and define for $l \in \{1, \ldots, m\}$, $j_l = \text{pos}_r(l, w)$, where $w$ is the string formed by the labels of the children of $n_0$. Let $(\pi_0, \ldots, \pi_{|\cdot|}; (R_d)_{d \in D})$ be the rule associated to the context $(p, a, j_k)$. Now define $W(a(n))$ as the sequence

$$\pi_0(n_0) \cdot \pi_{j_1}(n_1) \cdot \pi_{j_{k-1}}(n_{k-1}) \cdot \# \cdot \pi_{j_k}(n_k) \pi_{j_{k+1}}(n_{k+1}) \cdots \pi_{j_m}(n_m).$$

For each $d$, we denote the language $R_d$ associated to $a(n)$ by $R_d^{a(n)}$.

If $v$ is a valuation then define $v(W(a(n)))$ as the string obtained from $W(a(n))$ by replacing each $b(m)$ in $W(a(n))$ by $v(b(m))$. Note that the empty sequence is just replaced by the empty string.

We are now ready to define the semantics of an extended AG $F$ on a derivation tree.

**Definition 4.8.** Given an extended AG $F$ and a derivation tree $t$, we define a sequence of partial valuations $(\mathcal{F}_j(t))_{j \geq 0}$ as follows:
1. $\mathcal{F}_0(t)$ is the valuation that maps, for every node $n$, $\text{lab}(n)$ to the label of $n$ and is undefined everywhere else;
2. for $j > 0$, if $\mathcal{F}_{j-1}(t)$ is defined on all $b(m)$ occurring in $W(a(n))$ then

$$\mathcal{F}_j(t)(a(n)) = d,$$

where $\mathcal{F}_{j-1}(t)(W(a(n))) \in R_d^{a(n)}$. Note that this is well defined.
If for every $t$ there is an $l$ such that $\mathcal{F}_l(t)$ is totally defined (this implies that $\mathcal{F}_l(t) = \mathcal{F}_{l+1}(t)$) then we say that $\mathcal{F}$ is non-circular. Obviously, non-circularity is an important property. In the next section, we show that it is decidable whether an extended AG is non-circular. Therefore, we can state the following proviso.

**Proviso 4.9.** In the sequel we always assume an extended AG to be non-circular. Testing for non-circularity is discussed in the next section.

**Definition 4.10.** The valuation $\mathcal{F}(t)$ equals $\mathcal{F}_l(t)$ with $l$ such that $\mathcal{F}_l(t) = \mathcal{F}_{l+1}(t)$.

**Proviso 4.11.** Whenever we say query, we always mean unary query.

An extended AG $\mathcal{F}$ can be used in a simple way to express queries. Among the attributes in the vocabulary of $\mathcal{F}$, we designate some attribute $\text{result}$, and define:

**Definition 4.12.** An extended AG $\mathcal{F}$ with designated attribute $\text{result}$ expresses the query $Q$ defined by

$$Q(t) = \{ n \mid \mathcal{F}(t)(\text{result}(n)) = 1 \}$$

for every tree $t$.

**Example 4.13.** Recall the extended AG $\mathcal{F}$ of Fig. 1. This extended AG selects the first poem, all empty poems, and every poem that has the strings king or lord in every other verse starting from the first one. In Fig. 2 an illustration is given of the result of $\mathcal{F}$ on a derivation tree $t$. At each node $n$, we show the values $\mathcal{F}(t)(W(a(n)))$ and $\mathcal{F}(t)(a(n))$. We abbreviate $a(n)$ by $a$, king by $k$, lord by $l$, and king_lord by $k_l$.

The definition of the inherited attribute first indicates how the use of # can distinguish in a uniform way between different occurrences of the grammar symbol Poem. This is only a simple example. In the next section, we show that extended AGs can express all queries definable in MSO. Hence, they can also specify all relationships between siblings definable in MSO.

The language $R_1$ associated to result (cf. Fig. 1), contains those strings representing that the current Poem is the first one, or representing that it is not the first one and that it is either empty or that for every other verse starting at the first one the value of the attribute king_lord is 1.

**5. Non-circularity**

In this section, we show that it is decidable whether an extended AG is non-circular. In particular, we show that deciding non-circularity is in EXPTIME. As it is well known that deciding non-circularity of standard AGs is complete for EXPTIME [29], this result indicates that going from ranked to unranked trees does not increase the complexity of the non-circularity problem.

We first make the following remark indicating that testing non-circularity for extended AGs is slightly more subtle than for standard AGs.
Remark 5.1. Not all the specified attributes in a semantic rule are always used. Indeed, consider the grammar with productions $C \rightarrow A + B$, $A \rightarrow c$ and $B \rightarrow c$. Let $F$ be an extended AG where the inherited attribute $a$ of $A$ and $B$ is defined in the context $(C \rightarrow A + B, a, 1)$ as

$$a(1) := \langle \pi_0 = \varepsilon, \pi_1 = \varepsilon, \pi_2 = a; R_1 \rangle$$

and in the context $(C \rightarrow A + B, a, 2)$ as

$$a(2) := \langle \pi_0 = \varepsilon, \pi_1 = a, \pi_2 = \varepsilon; R_1 \rangle.$$

At first sight $F$ seems circular. This is, however, not the case since $A$ and $B$ never occur simultaneously in a derivation tree. Consider for example the tree $t = C(A(c))$. If the label of $n$ is $A$ then $W(a(n))$ is the empty sequence and consequently $F(t)(a(n)) = 1$ if and only if the empty string belongs to $R_1$.

We adopt an automata approach. To this end, we first generalize the tree walking automata of Bloem and Engelfriet [7] to unranked trees. In particular, we show that for each extended AG $F$, there exists a tree walking automaton $W_F$ such that $F$ is non-circular if and only if $W_F$ does not cycle on any input tree. Moreover, $W_F$ can be constructed in time polynomial in the size of $F$. We thus obtain our result by showing that testing whether a tree walking automaton cycles is in EXPTIME. At the end of the section, we briefly discuss a reduction to the circularity problem of standard AGs suggested by one of the referees.
Definition 5.2. A non-deterministic tree walking automaton is a tuple $W = (Q, \Sigma, \delta, q_0, F)$ where

- $Q$ is a finite set of states,
- $\Sigma$ is an alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states, and
- $\delta \subseteq Q \times (\Sigma \cup \{\downarrow, \uparrow, \leftarrow, \rightarrow\}) \times Q \times \{\downarrow_{\text{first}}, \downarrow_{\text{last}}, \leftarrow, \rightarrow, \uparrow, \text{stay}\}$ is the transition relation.

Intuitively, a tree walking automaton walks over the tree starting at the root. To make sure that the automaton cannot fall off the tree, we augment input trees with the boundary symbols $\leftarrow, \rightarrow, \downarrow,$ and $\uparrow$. For example, the tree $t := a(b, c)$ augmented with boundary symbols is defined as

$\text{bound}(t) := \downarrow (\rightarrow, a(\rightarrow, b(\uparrow), c(\uparrow), \leftarrow), \leftarrow),$

or more graphically:

$\downarrow \rightarrow a \leftarrow$

$\rightarrow b \quad \quad c \leftarrow$

$\uparrow \uparrow$

We use another auxiliary notion. We define $b(t)$ as $\text{bound}(t)$ without boundary symbols for root($t$). That is, $b(t)$ is the tree graphically represented by

$a$

$\rightarrow b \quad c \leftarrow$

$\uparrow \uparrow$

A perhaps more elegant solution is to have a separate transition function for the root node, internal nodes and leaf nodes. But since this last approach terribly complicates the proof of the next lemma we just stick to the tree representation with boundary symbols.

We still have to explain the semantics of a tree walking automaton. Depending on the current state and on the label at the current node, the transition relation determines in which direction the automaton can move and into which state it can change. The possible directions w.r.t. the current node are: go to the first child, the last child, the left sibling, the right sibling, or the parent, or stay at the current node. We have the straightforward restrictions that $W$ can only move to the left, right, down and up, when it reads the symbols $\leftarrow, \rightarrow, \downarrow,$ and $\uparrow$, respectively.

The automaton accepts an input tree when there exists a walk started at the root in the start state that again reaches the root node in a final state. We make this more precise. A configuration of $W$ on a tree $t$ is a pair $(n, q)$ where $n$ is a node of $\text{bound}(t)$ and $q \in Q$. The start configuration is $(\text{root}(t), q_0)$, and each $(\text{root}(t), q)$ with $q \in F$ is an accepting configuration. A walk of $W$ on $t$ is a (possibly infinite) sequence of configurations $c_1c_2c_3 \cdots$ such that each $c_{i+1}$ can be reached from $c_i$ by making one transition. The latter is defined in the obvious way. A walk is accepting when it is finite, its first configuration is the start configuration, and the last configuration is an accepting one. Finally, $W$ accepts $t$ when there exists an accepting walk of $W$ on $t$. However, we will not need this latter definition any further, as we are only interested in the existence of infinite walks.

We need the following definition to state the next lemma.
Definition 5.3. A non-deterministic tree walking automaton cycles if there is a tree on which it has an infinite walk starting in the start configuration.

Lemma 5.4. Deciding whether a non-deterministic tree walking automaton cycles, is in EXPTIME.

Proof. Let \( W = (Q, \Sigma, \delta, q_0, F) \) be a non-deterministic tree walking automaton. For a tree \( t \) define the behavior relation of \( W \) on \( t \) as the relation \( f_t^W \subseteq Q \times (Q \cup \{\#\}) \) as follows. For each \( q, q' \in Q \),

1. \( f_t^W(q, q') \) if there exists a walk of \( W \) starting at the root of \( t \) in state \( q \) that again returns at the root in state \( q' \) with the additional requirement that \( W \) does not move to the left sibling, the right sibling or the parent of the root (recall that these are labeled with \( \rightarrow, \leftarrow, \) and \( \downarrow \), respectively) during this walk; in brief, \( W \) walks only in \( b(t) \);
2. \( f_t^W(q, \#) \) if there is an infinite walk of \( W \) starting at the root in state \( q \), again with the additional requirement that \( W \) does not move to the left sibling, the right sibling or the parent of the root during this walk.

The additional requirement mentioned in both of the above cases is needed because we want to compute behavior relations of nodes in a tree, in terms of the behavior relations at the children of those nodes. Therefore, the behavior relations of the subtrees should only be defined by computations that do not leave those subtrees.

Let \( f \subseteq Q \times (Q \cup \{\#\}) \) be a relation and let \( \sigma \in \Sigma \). Then, we say that \( (f, \sigma) \) is satisfiable whenever there exists a tree \( t \) with \( f_t^W = f \) and the label of root(\( t \)) is \( \sigma \). We refer to the tuples \( (f, \sigma) \) as behavior tuples. It now suffices to compute the set of all satisfiable behavior tuples to decide whether \( W \) cycles. To see this, we first introduce the following relations that determine the behavior of \( W \) when it encounters the boundary of a tree at its root. Define the relations \( \delta_{\uparrow\downarrow}^\sigma, \delta_{\uparrow\leftarrow}^\sigma, \) and \( \delta_{\uparrow\rightarrow}^\sigma \), as follows: for each \( q, q' \in Q \),

- \( \delta_{\uparrow\downarrow}^\sigma(q, q') \) iff there exists \( q'' \) such that \( \delta(q, \sigma, q'', \rightarrow) \) and \( \delta(q'', \leftarrow, q', \leftarrow) \);
- \( \delta_{\uparrow\leftarrow}^\sigma(q, q') \) iff there exists \( q'' \) such that \( \delta(q, \sigma, q'', \leftarrow) \) and \( \delta(q'', \rightarrow, q', \rightarrow) \);
- \( \delta_{\uparrow\rightarrow}^\sigma(q, q') \) iff there exists \( q'' \) such that \( \delta(q, \sigma, q'', \uparrow) \) and \( \delta(q'', \downarrow, q', \downarrow \text{final}) \).

We define the directed graph \( G(f_1, \ldots, f_n) \) with nodes \( Q \cup \{\#\} \) and edges \( (q, q') \) such that \( f_i(q, q') \) for some \( i \in \{1, \ldots, n\} \). Then \( W \) cycles iff there is a satisfiable behavior tuple \( (f, \sigma) \) such that in \( G(f, \delta_{\uparrow\downarrow}^\sigma, \delta_{\uparrow\leftarrow}^\sigma, \delta_{\uparrow\rightarrow}^\sigma) \) there is a path from \( q_0 \) to \( \# \) or a path from \( q_0 \) to a cycle.

To reduce the complexity of our algorithm we make use of a weaker notion of satisfiability. We say that a behavior tuple \( (f, \sigma) \) is weakly satisfiable whenever there exists a satisfiable behavior tuple \( (g, \sigma) \) such that \( f \subseteq g \). Note that every satisfiable tuple is also weakly satisfiable. If \( g \) is witnessed by \( t \) then we say that \( f \) is weakly witnessed by \( t \). Further, let \( S \) be the set of all weakly satisfiable behavior tuples. Then \( W \) cycles whenever there exists a behavior tuple \( (f, \sigma) \in S \) such that in \( G(f, \delta_{\uparrow\downarrow}^\sigma, \delta_{\uparrow\leftarrow}^\sigma, \delta_{\uparrow\rightarrow}^\sigma) \) there is a path from \( q_0 \) to \( \# \) or a path from \( q_0 \) to a cycle.

In Fig. 3, we give an algorithm computing \( S \). In this algorithm, \( C \) is initialized by the set of weakly satisfiable behavior tuples witnessed by 1-node trees \( \sigma \). These are easily computed: just run the automaton on \( \sigma(\uparrow) \). Hereafter, the algorithm tests for each behavior tuple \( (f, \sigma) \) whether it can be obtained by combining behavior tuples in \( C \) and adds \( (f, \sigma) \) to \( C \) if this is the case. To this end, we use an automaton \( M_{f,\sigma} \) over the alphabet consisting of all behavior tuples. In particular, if \( (f_1, \sigma_1), \ldots, (f_n, \sigma_n) \) are weakly
satisfiable and \((f_1, \sigma_1) \cdots (f_n, \sigma_n) \in L(M, f, \sigma)\) then \((f, \sigma)\) is weakly satisfiable. Moreover, if each \((f_i, \sigma_i)\)

is weakly witnessed by \(t_i\), then \((f, \sigma)\) is weakly witnessed by \(\sigma(t_1, \ldots, t_n)\). From this it follows that all tuples in \(C\) are weakly satisfiable. The converse can be shown by induction on the minimal height of the trees weakly witnessing the weakly satisfiable behavior tuples. It follows that after completion

of the algorithm \(C = S\). Since the size of each \(M, f, \sigma\) will be exponential in the size of \(W\), the test \(L(M, f, \sigma) \cap C^* \neq \emptyset\) can be done in exponential time. As there are only exponentially many behavior tuples, the REPEAT loop will iterate at most an exponential number of times. Thus, the total execution time of the algorithm will be exponential in the size of \(W\).

It remains to explain the construction of \(M, f, \sigma\). First, we define a non-deterministic two-way string automaton \(M, f, \sigma\) with one pebble whose number of states is polynomial in the size of \(W\). By Proposition 2.9

and the comments following it, \(M'\) is equivalent to a one-way non-deterministic automaton whose size is only exponential in the size of \(M, f, \sigma\). We then define \(M, f, \sigma\) as the latter automaton and the proof is finished.

On input \((f_1, \sigma_1) \cdots (f_n, \sigma_n)\), \(M', f, \sigma\) works as follows. We only have to consider the case where each \((f_i, \sigma_i)\) is weakly satisfiable. Therefore, let each \((f_i, \sigma_i)\) be weakly witnessed by \(t_i\).

1. For each \(q, q' \in Q\) for which \(f(q, q')\), the automaton \(M', f, \sigma\) has to check whether there exists a walk starting at the root of \(\sigma(t_1, \ldots, t_n)\) in state \(q\) that again reaches the root in state \(q'\). However, \(M', f, \sigma\) does not need to know the tree \(\sigma(t_1, \ldots, t_n)\): \(M', f, \sigma\) just guesses this path using the \(f_i\)'s. That is, \(M', f, \sigma\) starts in state \(q\) at the root. If \(W\), for example, decides to move to the last child in state \(q_1\), then \(M', f, \sigma\) walks to the last position of the string \(\rightarrow (f_1, \sigma_1) \cdots (f_n, \sigma_n) \leftarrow\) arriving there in state \(q_1\). Further, if \(M', f, \sigma\) arrives at a position labeled with \((f_i, \sigma_i)\) and \(W\) decides to enter the subtree below this position, then \(M', f, \sigma\) just examines the relation \(f_i\) to see in which states it can return. If \(W\) makes a move to, say, the right sibling in state \(q_2\), then \(M', f, \sigma\) just makes a right move to state \(q_2\). If \(M', f, \sigma\) succeeds in reaching the root in state \(q'\), then it considers the next pair of states \(q_1\) and \(q_1'\) for which \(f(q_1, q_1')\). If all pairs are checked, \(M', f, \sigma\) moves to the next step.

2. For every \(q \in Q\) such that \(f(q, \#)\), \(M', f, \sigma\) has to verify the existence of an infinite walk on \(\sigma(t_1, \ldots, t_n)\) starting from state \(q\) at the root. This can happen in two ways. The first possibility is that \(W\) gets into a cycle in one of the subtrees \(t_1, \ldots, t_n\), say \(t_i\). This can be detected, like in the previous case, by simply guessing a walk reaching position \(i\) of the input string \((f_1, \sigma_1) \cdots (f_n, \sigma_n)\) in a state \(q'\) such that \(f_i(q', \#)\). The second possibility is that \(W\) can walk forever on the children of the root. We use the pebble to detect this: \(M', f, \sigma\) now just guesses a walk of \(W\) using the relations \(f_1, \ldots, f_n\) as explained above and non-deterministically
Theorem 5.5. Deciding non-circularity of extended AGs is in EXPTIME.

Proof. Let $F$ be an extended AG with attribute set $A$ and semantic domain $D$. For ease of exposition we assume that all grammar symbols have all attributes, i.e., for every $X \in N \cup T$, $\text{Inh}(X) \cup \text{Syn}(X) = A$.

We construct a tree walking automaton $W_F$ such that $W_F$ cycles if and only if $F$ is circular. Rather than letting $W_F$ work on derivation trees of $G$, we let it work on the set of all trees over the alphabet $(N \cup T) \times (P \cup T) \times (P \cup \{U\}) \times \{1, \ldots, m\}$ where $m = \max\{|r| \mid X \rightarrow r \in P\}$. That is, $m$ denotes the maximal number of positions of a regular expression in a production of $P$.

The automaton $W_F$ first checks the consistency of the labelings. That is, for each node $n$ of the input tree labeled with $(\sigma, p_1, p_2, i)$,

1. if $p_1 \in T$ then $\sigma = p_1$ and $n$ is a leaf; if $p_1 = X \rightarrow r \in P$ then $\sigma = X$ and $n$ is derived by $p_1$;
2. if $p_2 = U$ then $n$ is the root; if $p_2 \in P$ then the parent of $n$ is derived by $p_2$; and
3. if the parent $p$ of $n$ is derived by $X \rightarrow r$, $w$ is the string formed by the children of $p$, and $n$ is the $j$th child of $p$, then $\text{pos}_r(j, w) = i$.

The automaton checks this in the following way. It makes a depth first traversal of the tree. At each node $n$ labeled with $(\sigma, p_1, p_2, i)$ it can check (1) by first checking whether the current node is a leaf, and if not, by simulating the NFA $M_r$ of Lemma 2.3 on the children of $n$ where $p_1 = X \rightarrow r$. Only when the NFA

\[4\] The dependency graph $D_F(t)$ of $F$ for a derivation tree $t$ is defined as follows. Its nodes are all $a(n)$, such that $n$ is a node of $t$ and $a$ is an attribute of the label of $n$. Further, there is an edge from $a(n)$ to $b(m)$ if and only if $a(n)$ occurs in $W(b(m))$ (cf. Definition 4.7). Clearly, $F$ is well defined on $t$ if and only if $D_F(t)$ contains no cycle. Hence, $F$ is non-circular if and only if there does not exist a $t$ such that $D_F(t)$ is cyclic.
accepts it moves to the next node in the depth first traversal. To check (2), $W_F$ makes another depth first traversal of the tree. It first checks whether the root is labeled with $(U, p, U, i)$. Next, for each internal node $n$ labeled with $(a, p_1, p_2, i)$ it checks whether every child of $n$ has $p_1$ in the third component of its label. Finally, (3) is checked by making a third depth first traversal through the tree. Arriving at a node $n$ derived by $X \rightarrow r$, the automaton simulates $M_r$ on the children of $n$ to check that the fourth component of every label is equal to the state of $M_r$ after reading that label.

If all this succeeds then $W_F$ non-deterministically walks to a node and chooses an attribute $a$ which it keeps in its state. Now, suppose $W_F$ arrives at a node $n$ labeled with $(X, p_1, p_2, j)$ with the attribute $a$ in its state. When $a = \text{lab}$ or all $\pi_i$’s in the corresponding rule are empty then $W_F$ halts. Otherwise, we distinguish two cases.

1. $a$ is a synthesized attribute of $X$: Let $a(0) := \langle \pi_0, \ldots, \pi_{|\pi_r|}; (R_d)_{d \in D} \rangle$ be the rule in the context $(p_1, a, 0)$. Then $W_F$ non-deterministically chooses an attribute $b$ in a $\pi_i$ and replaces $a$ in its state with $b$. If $i = 0$ then $W_F$ just stays at the current node. If $i > 0$ then $W_F$ walks non-deterministically to a child of the current node having $i$ as the last component of its label.

2. $a$ is an inherited attribute of $X$: Let $a(j) := \langle \pi_0, \ldots, \pi_{|\pi_r|}; (R_d)_{d \in D} \rangle$ be the rule in the context $(p_2, a, j)$. Then $W_F$ non-deterministically chooses an attribute $b$ in a $\pi_i$ and replaces $a$ in its state with $b$. If $i = 0$ then $W_F$ walks to the parent of $n$. If $i > 0$ then $W_F$ walks non-deterministically to a sibling of $n$ having $i$ as the last component of its label (or possibly stays at $n$ if $i = j$).

Note that $W_F$ also halts in case such non-deterministic choices do not exist. Clearly, $W_F$ cycles if and only if $\mathcal{F}$ is circular. Moreover, $W_F$ can be constructed in time polynomial in the size of $\mathcal{F}$. The theorem then follows from Lemma 5.4.

Since deciding non-circularity for standard attribute grammars is also hard for EXPTIME, we obtain that testing whether a non-deterministic tree walking automaton cycles is EXPTIME-complete.

**Remark 5.6.** One of the referees pointed out that Theorem 5.5 can also be obtained by a reduction to the circularity problem of standard attribute grammars. Though the latter reduction, as sketched below, is simpler than the automata proof, we chose to keep our proof as we get as a byproduct that detecting cycles in tree-walking automata is in EXPTIME. This complexity might be useful in XML research as tree-walking automata seem to belong to the XML research toolkit [39,40,47].

We sketch the reduction to the circularity problem of standard AGs. For every production $p : X \rightarrow r$, let the states of the NFA $M_r$ be called $(p, k)$ where $k$ is a position in $r$. Now, define a standard CFG which has the same non-terminals as the given extended AG, plus all non-terminals $(p, k)$ with $(p, 0)$ identified with $X$. The productions of this CFG are obtained from the automata $M_r$ in the usual way, as in the simulation of finite automata by right-linear grammars. Thus, in every derivation tree, an application of $p$ is replaced by a right-linear piece of binary tree as usual. It remains to add semantic rules (or at least dependencies) to this CFG in such a way that circularity carries over.

For instance, to “implement” the definition of a synthesized attribute $a$ of $X$, add $a$ as a synthesized attribute to every $(p, k)$. For a production $(p, k) \rightarrow Y(p, m)$ of the CFG, the attribute $a$ of $(p, k)$ depends on the attribute $a$ of $(p, m)$ and on the $\pi_m$ attributes in the rule for $a(0)$ in $p$ (and, moreover on the $\pi_0$ attributes of that rule in the case that $k = 0$ and $(p, k) = X$).

It is slightly more difficult to implement the definition of an inherited attribute $b$ of a non-terminal $Z$ at a position $j$ in $r$. To do so, add an inherited attribute (inh, $b$, $j$) and a synthesized attribute (syn, $b$, $j$) to
every \((p, k), k > 0\). For a production, \((p, k) \rightarrow Y(p, m)\) of the CFG with \(k > 0\), the attribute \((\text{syn}, b, j)\) of \((p, k)\) depends on the attribute \((\text{syn}, b, j)\) of \((p, m)\) and on the \(\pi_m\) attributes of the semantic rule for \(b(j)\) in \(p\); and, the inherited attribute \((\text{inh}, b, j)\) of \((p, m)\) depends on the attribute \((\text{inh}, b, j)\) of \((p, k)\) and on the \(\pi_m\) attributes of the semantic rule for \(b(j)\) in \(p\). For \(k = 0\) (and so \((p, k) = X\)), the attribute \((\text{inh}, b, j)\) of \((p, m)\) depends on the \(\pi_m\) and \(\pi_0\) attributes of the semantic rule for \(b(j)\) in \(p\). Moreover, if \(Y = Z\) and \(m = j\), then the attribute \(b\) of \(Y\) depends on the attribute \((\text{inh}, b, j)\) of \((p, k)\) (if \(k > 0\)) and on the attribute \((\text{syn}, b, j)\) of \((p, m)\). It can be shown that the standard AG is circular iff the extended AG is. □

6. Optimization

An important research topic in the theory of query languages is that of optimization of queries. This comprises, for example, the detection and elimination of subqueries that always return the empty relation, or more general, the rewriting of queries, stated in a certain formalism, into equivalent ones that can be evaluated more efficiently. The central problem in the case of the latter is, hence, to decide whether the rewritten queries are indeed equivalent to the original ones. In this section, we study the complexity of the emptiness and equivalence test of extended AGs. Interestingly, these results will be applied in the next section to obtain a new upper bound for deciding equivalence of Region Algebra expressions introduced by Consens and Milo [16].

We consider the following problems:

- **Non-emptiness**: Given an extended AG \(\mathcal{F}\), does there exists a tree \(t\) and a node \(n\) of \(t\) such that \(\mathcal{F}(t)(\text{result}(n)) = 1\)?

- **Equivalence**: Given two extended AGs \(\mathcal{F}_1\) and \(\mathcal{F}_2\) over the same ECFG, do \(\mathcal{F}_1\) and \(\mathcal{F}_2\) express the same query?

To show the EXPTIME-hardness for the above decision problems we use a reduction from universality of NBTAs. Recall that NBTAs are defined in Section 2.4. The matching upper bound is obtained by a reduction to the emptiness problem of NBTAs.

A binary NFTA is an NFTA where the length of every string in a transition relation is either 2 or 0. Such automata, hence, work on binary trees. Seidl showed that already for such automata it is EXPTIME-hard to determine whether they accept all trees [48].

**Lemma 6.1.** Given a binary NFTA \(B\), it is EXPTIME-complete to test whether \(B\) accepts all trees.

**Lemma 6.2.** Deciding non-emptiness of extended AGs is hard for EXPTIME.

**Proof.** Let \(B = (Q, \Sigma, F, \delta)\) be a binary NFTA. We construct an AG \(\mathcal{F}\) such that \(\mathcal{F}\) is non-empty iff \(B\) rejects a binary tree. The theorem then follows from Lemma 6.1.

Roughly speaking, \(\mathcal{F}\) is an AG over a ECFG \(G\) defining all binary \(\Sigma\)-trees; \(Q \cup \{\text{result}\} \subseteq A; \mathcal{F}(t)(q(n)) = 1\) iff \(q \in \delta^*(t_n)\) and \(\mathcal{F}(t)(\text{result}(n)) = 1\) iff \(\delta^*(t_n) \cap F = \emptyset\).

Formally, \(G = (N, T, P, S)\) where \(N = \{S\} \cup \{S_\sigma \mid \sigma \in \Sigma\}, T = \Sigma,\) and \(P = \{S_\sigma \rightarrow (S_{\sigma_1} + \cdots + S_{\sigma_k})| S_{\sigma_1} + \cdots + S_{\sigma_k} + \sigma \mid \sigma \in \Sigma\} \cup \{S \rightarrow S_{\sigma_1} + \cdots + S_{\sigma_k} \mid \sigma \in \Sigma\}\) where \(\Sigma = \{\sigma_1, \ldots, \sigma_k\}\).
Let \( Q = \{q_1, \ldots, q_n\} \). Then, \( A = Q \cup \{\text{result}, \text{lab}\} \cup \Gamma \), where \( \Gamma = \{\gamma_{i,j} \mid i, j \in \{1, \ldots, n\}\} \). All attributes are synthesized. For every non-terminal \( X \neq S \), \( \text{Syn}(X) = Q \cup \Gamma \); \( \text{Syn}(S) = \{\text{result}\} \). An attribute \( \gamma_{i,j} \) is true for a node \( n \) iff both the attribute \( q_i \) and \( q_j \) are true for \( n_1 \) and \( n_2 \), respectively. So, \( q \) is true for \( n \) iff at least one \( \gamma_{i,j} \) is true for which \( q_i q_j \in \delta(q, \sigma) \) where \( n \) is labeled with \( \sigma \). Recall that we are working with binary trees. The semantic domain is \( D = \{0, 1\} \cup \mathbb{N} \cup T \).

For \( i, j \in \{1, \ldots, n\} \) and \( q \in Q \), define the following rules (as usual, unspecified \( R_d \)'s are empty):

1. In context \( (S_\sigma \rightarrow (S_{\sigma_1} + \cdots + S_{\sigma_k})(S_{\sigma_1} + \cdots + S_{\sigma_k}) + \sigma, \gamma_{i,j}, 0) \), we have
   \[
   (\pi_0 = \varepsilon, \pi_1 = (q_1, \ldots, q_n), \ldots, \pi_{2k} = (q_1, \ldots, q_n), \pi_{2k+1} = \varepsilon; R_1, R_0)
   \]
   where \( R_1 = \{w_1 \cdots w_n w'_1 \cdots w'_n \mid w_i = 1, w'_j = 1\} \) and \( R_0 \) is the complement of \( R_1 \) w.r.t. \( D^* \). Clearly, \( R_1 \) and \( R_0 \) can be recognized by DFA's of size polynomial in the size of \( B \).

2. In context \( (S_\sigma \rightarrow (S_{\sigma_1} + \cdots + S_{\sigma_k})(S_{\sigma_1} + \cdots + S_{\sigma_k}) + \sigma, q, 0) \), we have
   \[
   (\pi_0 = \langle \gamma_{1,1}, \gamma_{1,2}, \ldots, \gamma_{1,n}, \gamma_{2,1}, \ldots \rangle, \pi_1 = \varepsilon, \ldots, \pi_{2k} = \varepsilon, \pi_{2k+1} = \text{lab}; R_1, R_0).
   \]
   It remains to define \( R_1 \) and \( R_0 \). Let \( V_{q,\sigma} \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\} \) be those pairs \((i, j)\) such that \( q_i q_j \in \delta(q, \sigma) \). Then \( R_1' \) contains precisely those strings \( w \) of length \( n^2 \) such that the \(((i-1)\times n+j)\)th position is 1 for at least one \((i, j) \in V_{q,\sigma}' \). Let \( R_0'' = \{\sigma\} \) if \( \varepsilon \in \delta(q, \sigma) \) and \( R_0'' = \emptyset \) otherwise. Then \( R_1 = R_1' \cup R_1'' \) and \( R_0 = D^* - R_1 \). Again, \( R_1 \) and \( R_0 \) can be recognized by DFA’s of size polynomial in the size of \( B \).

3. In context \( (S \rightarrow S_{\sigma_1} + \cdots + S_{\sigma_k}, \text{result}, 0) \), we have
   \[
   (\pi_0 = \varepsilon, \pi_1 = (q_1, \ldots, q_n), \ldots, \pi_k = (q_1, \ldots, q_n); R_1, R_0)
   \]
   where \( R_1 \) contains precisely those strings \( w_1 \cdots w_n \) where \( w_i = 0 \) if \( q_i \in F \); \( R_0 \) is the complement of \( R_1 \) w.r.t. \( D^* \). Clearly, \( R_1 \) and \( R_0 \) can be recognized by DFA’s of size polynomial in the size of \( B \).

For a binary \( \Sigma \)-tree \( t \), let der(\( t \)) be the derivation tree of \( G \) corresponding to \( t \). It is not difficult to see that the root of der(\( t \)) is selected by \( F \) iff no run of \( B \) accepts \( t \). Hence, \( F \) is non-empty iff \( B \) rejects a binary tree.

Non-emptiness of extended AGs can in fact also be decided in \( \text{EXPTIME} \). The proof essentially works as follows. For each extended AG \( F \) we construct an NBTA \( T_F \) guessing the attribute values at each node; it then accepts when the result attribute of at least one node is true. Since the size of \( T_F \) will be exponential in the size of \( F \) and non-emptiness of NBTA’s can be checked in polynomial time (see Lemma 2.8), we obtain an \( \text{EXPTIME} \) algorithm for testing non-emptiness of extended AGs.

Bloem and Engelfriet [8] already showed that tree automata can guess attribute values of nodes defined by standard finite-valued attribute grammars which have ranked derivation trees. We must extend this technique to unranked trees and automata, and must control the sizes of the NFAs involved in the transition function of the automaton. In particular, we control these sizes by first describing the transition function by non-deterministic two-way automata with a pebble which can be transformed into equivalent one-way non-deterministic automata with only an exponential size increase.

**Theorem 6.3.** Deciding non-emptiness of extended AGs is \( \text{EXPTIME}-\text{complete} \).
Proof. EXPTIME-hardness has just been shown in Lemma 6.2, so it remains to show that non-emptiness is in EXPTIME.

Let \( F \) be an extended AG over the grammar \( G = (N, T, P, U) \). Recall that all regular languages \( R_d \) are represented by NFAs. W.l.o.g., we assume that every grammar symbol has all attributes, i.e., for all \( X \in N \cup T \), \( \text{Inh}(X) \cup \text{Syn}(X) = A \). As mentioned above, we construct an NBTA \( T_F \) such that \( L(T_F) \cap L(G) \neq \emptyset \) if and only if \( F \) is non-empty. The size of \( T_F \) will be exponential in the size of \( F \). That is, the set of states of \( T_F \) and the NFAs representing transition functions will be exponential in the size of \( F \). By Lemmas 2.8 and 2.7, non-emptiness of \( L(T_F) \cap L(G) \) can be checked in time exponential in the size of \( F \). Hence, the theorem follows.

We say that an arbitrary total valuation \( v \) of \( t \) satisfies \( F \) if for every node \( n \) of \( t \) and attribute \( a \) of the label of \( n \), \( v(W(a(n))) \in R^a_{v(a(n))} \). It follows immediately from the definitions that \( F(t) \) satisfies \( F \). Moreover, \( F(t) \) is the only valuation that satisfies \( F \). Indeed, suppose that \( v \) satisfies \( F \). An easy induction on \( l \), using non-circularity, then shows that if \( a(n) \) is defined in \( F(t) \) then \( F_i(t)(a(n)) = v(a(n)) \).

The automaton \( T_F = (Q, N \cup T, F, \delta) \) essentially guesses the values of the attributes and then verifies whether they satisfy all semantic rules (i.e., whether the resulting valuation satisfies \( F \)). Therefore, we use as set of states \( Q \) all tuples \( (x, o, p, i) \) where \( x : A \to D \) is a function, \( o \in \{0, 1\} \), \( p \in P \cup T \) and \( i \in \{1, \ldots, m\} \), where \( m = \max\{|r| \mid X \to r \in P\} \).

The intended meaning of the states is as follows. If a valid state assignment (cf. Section 2.4) of \( T_F \) assigns the state \( q = (x, o, p, i) \) to a node \( n \) of an input tree \( t \), then

- \( x \) represents the values of the attributes of \( n \); i.e., for all \( a \in A \), \( F(t)(a(n)) = x(a) \);
- \( o = 1 \) if and only if a node in \( t_n \) has been selected;
- if \( n \) is labeled with a non-terminal then \( p \in P \) and \( n \) is derived by \( p \); otherwise \( p \in T \) and \( n \) is labeled by \( p \); and
- if the parent \( p \) of \( n \) is derived by \( X \to r \), \( n \) is the \( j \)th child of \( p \), and \( w \) is the string formed by the labels of the children of \( p \), then \( \text{pos}_r(j, w) = i \).

For a tuple \( q = (x, o, p, i) \in Q \), we denote \( o \) by \( q_o \), \( x \) by \( q_x \), \( p \) by \( q_p \), and \( i \) by \( q_i \). If \( q_p \to_p X \to r \in P \) then we denote \( X \) by \( q.X \) and \( r \) by \( q.r \), and if \( p \in T \) then we denote \( p \) also by \( q.X \). If \( x : A \to D \) is a function and \( \pi = a_1 \cdots a_n \) is a sequence of attributes then we denote the string \( x(a_1) \cdots x(a_n) \) by \( x(\pi) \).

The set of final states \( F \) is defined as \( \{q \in Q \mid q_o = 1\} \). We define the transition function. For all \( x : A \to D \), \( \sigma_1, \sigma_2 \in T \), \( o \in \{0, 1\} \), and \( i \in \{1, \ldots, m\} \), define

\[
\delta((x, o, \sigma_1, i), \sigma_2) := \begin{cases} 
\{\varepsilon\} & \text{if } \sigma_1 = \sigma_2 \text{ and } o = x(\text{result}), \\
\emptyset & \text{otherwise.}
\end{cases}
\]

For all \( X \in N \) and \( q \in Q \), if \( q.X \neq X \) then \( \delta(q, X) = \emptyset \); otherwise, if \( q.X = X \) then \( q_1 \cdots q_n \in \delta(q, X) \) iff

1. \( q_1.X \cdots q_n.X \in L(q.r) \);
2. for all \( j = 1, \ldots, n \), \( \text{pos}_{q,r}(j, q_1.X \cdots q_n.X) = q_j.i \);
3. for every synthesized attribute \( a \) of \( X \), defined by the rule

\[
\langle \pi_0, \ldots, \pi_{|q.r|}; (R_d)_{d \in D}\rangle,
\]
in context \((q.p, a, 0)\), we must have
\[
q.\mathcal{A}(\pi_0) \cdot q_1.\mathcal{A}(\pi_{q_1,i}) \cdots q_n.\mathcal{A}(\pi_{q_n,i}) \in R_{q.\mathcal{A}(a)},
\]
for all \(j = 1, \ldots, n\), for every inherited attribute \(a\) of \(q_j.X\), defined by the rule
\[
(\pi_0, \ldots, \pi_{[q,r]}; (R_d)_{d\in D}),
\]
in context \((q.p, a, q_j.i)\), we must have
\[
q.\mathcal{A}(\pi_0) \cdot q_1.\mathcal{A}(\pi_{q_1,i}) \cdots q_{j-1}.\mathcal{A}(\pi_{q_{j-1},i}) \# q_j.\mathcal{A}(\pi_{q_j,i}) \quad q_{j+1}.\mathcal{A}(\pi_{q_{j+1},i}) \cdots q_n.\mathcal{A}(\pi_{q_n,i}) \in R_{q_j.\mathcal{A}(a)},
\]
and

5. \(q.o = 1\) if and only if \(q.\mathcal{A}(\text{result}) = 1\) or there exists a \(j \in \{1, \ldots, n\}\) such that \(q_j.o = 1\).

We show that conditions (1–5) are regular. Moreover, they can be defined by NFAs whose size is exponential in the size of \(F\). The result then follows since the size of the NFA computing the intersection of a constant number of NFAs is polynomial in the sizes of those NFAs.

- (1) and (2) are checked by the NFA \(M_{q.r}\) obtained from \(q.r\) as described in Lemma 2.3 whose size is polynomial in \(r\).
- For (3), we describe a two-way non-deterministic automaton \(M_1\). By Proposition 2.9, \(M_1\) can be transformed into an equivalent one-way NFA whose size is exponential in \(M_1\). \(M_1\) makes one pass through the input string for every synthesized attribute \(a\) of \(X\) simulating the NFA for \(R_{q.\mathcal{A}(a)}\). If the latter accepts then \(M_1\) walks back to the beginning of the input string and treats the next synthesized attribute or accepts if all synthesized attributes have been accounted for; \(M_1\) rejects if the NFA for \(R_{q.\mathcal{A}(a)}\) rejects. This needs only a linear number of states in the sizes of the NFAs representing transition functions and the set of attributes.
- For (4), we describe a two-way non-deterministic automaton \(M_2\) with a pebble. By Proposition 2.9, \(M_2\) can be transformed into an equivalent one-way NFA whose size is exponential in \(M_2\). \(M_2\) successively puts its pebble on each position of the input string. Suppose \(M_2\) has just put the pebble on position \(j\), then, for every inherited attribute \(a\) of \(q_j.X\), \(M_2\) walks back to the beginning of the input string and simulates the NFA for \(R_{q_j.\mathcal{A}(a)}\), pretending to read \(#\) the moment it encounters the pebble. If the NFA for \(R_{q_j.\mathcal{A}(a)}\) accepts, then \(M_2\) walks back to the beginning of the input string and treats the next inherited attribute of \(q_j.X\) or, if all inherited attributes of \(q_j.X\) have been considered, moves the pebble to position \(j + 1\) and repeats the same procedure. If \(M_2\) has put its pebble on all positions it accepts. This needs only a number of states linear in the size of the NFAs representing transition functions and the set of attributes.
- (5) can be done by making one pass over the input string using a constant number of states.

This concludes the proof of the theorem.

Let us now turn to the equivalence problem. This problem is actually polynomial-time equivalent to the complement of the non-emptiness problem (i.e., the emptiness problem), and hence it is also EXPTIME-complete. Indeed, \(F\) expresses the constant empty query if and only if it is equivalent to a trivial extended AG that expresses this query, and conversely, we can easily test if \(F_1\) and \(F_2\) express the same query by constructing an extended AG that first runs \(F_1\) and \(F_2\) independently, and then defines the value of
result of a node to be 0 iff the values of result for $F_1$ and $F_2$ on that node agree. This gives the following theorem.

**Theorem 6.4.** Deciding equivalence of extended AGs over the same ECFG is EXPTIME-complete.

### 7. Optimization of Region Algebra expressions

The region algebra introduced by Consens and Milo [16,15] is a set-at-a-time algebra, based on the PAT algebra [46], for manipulating text regions. In this section, we show that any Region Algebra expression can be simulated by an extended AG of polynomial size. This then leads to an EXPTIME algorithm for the equivalence and emptiness test of Region Algebra expressions. The algorithm of Consens and Milo is based on the equivalence test for first-order logic formulas over trees which has a non-elementary lower bound and, therefore, conceals the real upper bound of the former problem. Our algorithm drastically improves the complexity of the equivalence test for the Region Algebra and matches more closely the coNP lower bound [16].

It should be pointed out that our definition differs slightly from the one in [16]. Indeed, we restrict ourselves to regular languages as patterns, while Consens and Milo do not use a particular pattern language. This is no loss of generality since

- regular languages are the most commonly used pattern language in the context of document databases; and
- the huge complexity of the algorithm of [16] is not due to the pattern language at hand, but is due to quantifier alternation of the resulting first-order logic formula, induced by combinations of the operators ‘−’ (difference) and $<$, $>$, $\subset$, and $\supset$.

**Definition 7.1.** A region index schema $I = (S_1, \ldots, S_n, \Sigma)$ consists of a set of region names $S_1, \ldots, S_n$ and a finite alphabet $\Sigma$.

If $N$ is a natural number, then a region over $N$ is a pair $(i, j)$ with $i \leq j$ and $i, j \in \{1, \ldots, N\}$.

An instance $I$ of a region index schema $I$ consists of a string $I(\omega) = a_1 \ldots a_{N_I} \in \Sigma^+$ with $N_I > 0$, and a mapping (also denoted by $I$) associating to each region name $S_i$ a set of regions over $N_I$.

We abbreviate $r \in \bigcup_{i=1}^n I(S_i)$ by $r \in I$. We use the notation $L(r)$ (respectively, $R(r)$) to denote the location of the left (respectively, right) endpoint of a region $r$ and denote by $\omega(r)$ the string $a_{L(r)} \ldots a_{R(r)}$.

**Example 7.2.** Consider the region index schema $I = (\text{Proc}, \text{Func}, \text{Var}, \{a, b, c, \ldots, z\})$. In Fig. 4, an example of an instance over $I$ is depicted. Here, $N_I = 16$, $I(\omega) = \text{abcdefghijklmnp}$, $I(\text{Proc}) = \{(1, 16), (6, 10)\}$, $I(\text{Func}) = \{(12, 16)\}$ and $I(\text{Var}) = \{(2, 3), (6, 7), (12, 13)\}$.

For two regions $r$ and $s$ in $I$ define:

- $r < s$ if $R(r) < L(s)$ (r precedes s); and
- $r \subset s$ if $L(s) \leq L(r)$, $R(r) \leq R(s)$, and $s \neq r$ (r is included in s).

We also allow the dual predicates $r > s$ and $r \supset s$ which have the obvious meaning.
**Definition 7.3.** An instance $I$ is **hierarchical** if
- $I(S) \cap I(S') = \emptyset$ for all region names $S$ and $S'$ in $\mathcal{I}$, and
- for all $r, s \in I$ with $r \neq s$, one of the following holds: $r < s$, $s < r$, $r \subset s$ or $s \subset r$.

The last condition simply says that if two regions overlap then one is strictly contained in the other.

The instance in Fig. 4 is hierarchical. Like in [16], we only consider hierarchical instances. We now define the Region Algebra.

**Definition 7.4.** Region Algebra expressions over $\mathcal{I} = (S_1, \ldots, S_n, \Sigma)$ are inductively defined as follows:
- every region name of $\mathcal{I}$ is a Region Algebra expression;
- if $e_1$ and $e_2$ are Region Algebra expressions then $e_1 \cup e_2$, $e_1 - e_2$, $e_1 \subset e_2$, $e_1 < e_2$, $e_1 \supset e_2$, and $e_1 > e_2$ are also Region Algebra expressions;
- if $e$ is a Region Algebra expression and $R$ is a regular language over $\Sigma$ then $\sigma_R(e)$ is a Region Algebra expression.

The semantics of a Region Algebra expression on an instance $I$ is defined as follows:

$$
[S]^I = I(S);
[\sigma_R(e)]^I := \{ r \mid r \in [e]^I \text{ and } \omega(r) \in R \};
[e_1 \cup e_2]^I := [e_1]^I \cup [e_2]^I;
[e_1 - e_2]^I := [e_1]^I - [e_2]^I;
$$

and for $\star \in \{<, >, \subset, \supset\}$:

$$
[e_1 \star e_2]^I := \{ r \mid r \in [e_1]^I \text{ and } \exists s \in [e_2]^I \text{ such that } r \star s \}.
$$

We represent the regular languages occurring as patterns in Region Algebra expressions by DFAs. The size of a region algebra expression is the number of symbols plus the sizes of the DFAs for the regular languages occurring in it. The use of DFAs is crucial for our proof of Theorem 7.8: the use of NFAs would force us to apply the well-known subset construction to them, giving rise to an additional exponential in the time complexity of the algorithm. Nevertheless, we feel that in practice many patterns can already be represented by DFAs.

**Example 7.5.** The Region Algebra expression $\text{Proc} \supset \sigma_{\Sigma^* \text{start}} \Sigma^*(\text{Proc})$ defines all the Proc regions which contain a Proc region that contains the string start.
An important observation is that for any region index schema \( \mathcal{I} = (S_1, \ldots, S_n, \Sigma) \) there exists an ECFG \( G_{\mathcal{I}} \) such that any hierarchical instance of \( \mathcal{I} \) ‘corresponds’ to a derivation tree of \( G_{\mathcal{I}} \). This ECFG is defined as follows: \( G_{\mathcal{I}} = (N, T, P, U) \), with \( N = \{U, S_1, \ldots, S_n\} \), \( T = \Sigma \), and where \( P \) consists of the rules

\[
\begin{align*}
p_0 &:= U \rightarrow (S_1 + \ldots + S_n + \sigma_1 + \ldots + \sigma_k)^*; \\
p_1 &:= S_1 \rightarrow (S_1 + \ldots + S_n + \sigma_1 + \ldots + \sigma_k)^*; \\
&\vdots \\
p_n &:= S_n \rightarrow (S_1 + \ldots + S_n + \sigma_1 + \ldots + \sigma_k)^*.
\end{align*}
\]

Here, we assume \( \Sigma = \{\sigma_1, \ldots, \sigma_k\} \).

For example, the derivation tree \( t_I \) of \( G_{\mathcal{I}} \) representing the instance \( I \) of Fig. 4 is depicted in Fig. 5. Regions in \( I \) then correspond to nodes in \( t_I \) in the obvious way. We denote the node in \( t_I \) that corresponds to the region \( r \) by \( n_r \).

Since extended AGs can store results of subcomputations in their attributes, they are naturally closed under composition. It is, hence, no surprise that the translation of Region Algebra expressions into extended AGs proceeds by induction on the structure of the former.

**Lemma 7.6.** For every Region Algebra expression \( e \) over \( \mathcal{I} \) there exists an extended AG \( \mathcal{F}_e \) over \( G_{\mathcal{I}} \) such that for every hierarchical instance \( I \) and region \( r \in I, r \in [e]^I \) if and only if \( \mathcal{F}_e(t_I)(\text{result}_e(n_r)) = 1 \). Moreover, \( \mathcal{F}_e \) can be constructed in time polynomial in the size of \( e \).

**Proof.** The proof proceeds by induction on the structure of Region Algebra expressions. The extended AG \( \mathcal{F}_e \) will always contain the attribute \( \text{result}_e \) which is synthesized for all region names. As before the \( R_d \)’s that are not specified are assumed to be empty. Region Algebra expressions can only select regions, therefore, no attributes are defined for terminals. For the same reason, \( U \) has no attributes.

1. \( e = S_j; A_e = \{\text{result}_e, \text{lab}\}; D_e = \{0, 1, S_1, \ldots, S_n, U\} \cup \Sigma \); for \( i = 1, \ldots, n \), define in the context \( (p_i, \text{result}_e, 0) \) the rule

\[
\text{result}_e(0) := \langle \pi_0 = \text{lab}, \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; R_1 = \{S_j\}, R_0 = D_e^* - R_1 \rangle.
\]
2. \( e = \sigma_R(e_1) \): let \( M = (S, \Sigma, \delta, s_0, F) \) be the DFA accepting \( R \) with \( S = \{s_0, \ldots, s_m\} \). Define \( A_e = A_{e_1} \cup S \) and \( D_e = D_{e_1} \cup S \). W.l.o.g, we assume \( S \cap A_{e_1} = \emptyset \) and \( S \cap D_{e_1} = \emptyset \). We define the semantic rules of \( F_e \) as the semantic rules of \( F_{e_1} \) extended with the ones we describe next.

Each non-terminal has the synthesized attributes \( s_0, \ldots, s_m \). They are defined in \( F_e \) such that for a region instance \( I \) and region \( r \in I, F_e(t_I)(s(n_r)) = s' \) if and only if \( \delta^*(s, \omega(r)) = s' \). Observe that the attribute values together form the state transition function \( S \rightarrow S \) of the string \( \omega(r) \). So, for \( i = 1, \ldots, n \) and \( j = 1, \ldots, m \), define in the context \((p_i, s_j, 0)\) the rule

\[
s_j(0) := \langle \pi_0 = \varepsilon, \pi_1 = (s_0, \ldots, s_m), \ldots, \pi_n = (s_0, \ldots, s_m), \\
\pi_{n+1} = \text{lab}, \ldots, \pi_{n+k} = \text{lab}; (R^i_j)_{s \in S}\rangle.
\]

It remains to define the regular languages \( (R^i_j)_{s \in S} \). Note that the input strings for each \( R^i_j \) are of the form \( \bar{w} = w_1 \cdots w_k \), where for \( l = 1, \ldots, k, w_l \in S^{m+1} \) or \( w_l \in \Sigma \). The DFA \( M_{j,s} \) accepting \( R^i_j \) then works as follows: it starts in state \( s_j \), if \( w_1 \in S^{m+1} \) then \( M_{j,s} \) continues in state \( s' \) where \( s' \) occurs on the \((j + 1)\)th position of \( w_1 \) (this is the value of the attribute \( s_j \)); otherwise, if \( w_1 \in \Sigma \) then \( M_{j,s} \) continues in state \( \delta(s_j, w_1) \). Formally, \( M_{j,s} \) accepts \( \bar{w} \) if there exist \( j_0, j_1, \ldots, j_k \in \{0, \ldots, m\} \) such that

- \( j_0 = j \);
- for \( l = 2, \ldots, k, \) if \( w_l \in S^{m+1} \) then \( s_{j_l} \) is the \((j_l-1)\)th element of \( w_l \); if \( w_l \in \Sigma \) then \( s_{j_l} = \delta(s_{j_{l-1}}, w_l) \); and
- \( s_{j_k} = s \).

Clearly, \( M_{j,s} \) can be defined using a number of states polynomial in the size of \( S \). The attribute result\(_e\), then becomes true for a node \( n \), when \( F_e(t_I)(s_0(n)) \in F \) and \( F_e(t_I)(\text{result}_{e_1}(n)) = 1 \). So, for \( i = 1, \ldots, n \), define in the context \((p_i, \text{result}_e, 0)\) the rule

\[
\text{result}_e(0) := \langle \pi_0 = (s_0, \text{result}_{e_1}), \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; \\
R_1 = \{s \mid s \in F\}, R_0 = D^*_e - R_1 \rangle.
\]

In the following \( e \) will always depend on subexpressions \( e_1 \) and \( e_2 \). Hence, \( F_e \) always consist of \( F_{e_1} \) and \( F_{e_2} \) extended with rules for the new attributes. We, therefore, only specify the new rules. Also, we implicitly take \( D_e = D_{e_1} \cup D_{e_2} \). As we enlarge the domain, the old rules coming from \( F_{e_1} \) and \( F_{e_2} \) might not be total anymore. A solution is to add a dummy attribute value \( d_0 \) to \( D_e \) and set \( R_{d_0} = D^*_e - D^*_{e_1} \) in the rules of \( F_e \). We will always assume that (apart from the attribute lab) \( A_{e_1} \), \( A_{e_2} \) and the set of new attributes are disjoint.

3. \( e = e_1 \cup e_2 \): a node \( n \) is selected when \( F_e(t_I)(\text{result}_{e_1}(n)) = 1 \) or \( F_e(t_I)(\text{result}_{e_2}(n)) = 1 \). Define \( A_e = A_{e_1} \cup A_{e_2} \cup \{\text{result}_e\} \). So, for \( i = 1, \ldots, n \), define in the context \((p_i, \text{result}_e, 0)\) the rule

\[
\text{result}_e(0) := \langle \pi_0 = (\text{result}_{e_1}, \text{result}_{e_2}), \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; \\
R_1 = \{01, 10, 11\}, R_0 = D^*_e - R_1 \rangle.
\]

4. \( e = e_1 - e_2 \): similar to the previous case with \( R_1 = \{10\} \).

5. \( e = e_1 \supset e_2 \): define \( A_e = A_{e_1} \cup A_{e_2} \cup \{\text{down}, \text{result}_e\} \). Each region name has the synthesized attribute down such that for a region instance \( I \) and a region \( r, F_e(t_I)(\text{down}(n_r)) = 1 \) if there exists a
region $s$ such that $r \supset s$ and $s \in \llbracket e_2 \rrbracket$. So, for $i = 1, \ldots, n$, define in the context $(p_i, \text{down}, 0)$ the rule

\[
down(0) := (\pi_0 = \varepsilon, \pi_1 = (\text{result}_{e_2}, \text{down}), \ldots, \pi_n = (\text{result}_{e_2}, \text{down}),
\]

\[
\pi_{n+1} = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; R_1, R_0 = D_e^* - R_1,
\]

where $R_1$ is the regular language that contains all strings containing at least one 1. A node $n$ is then selected when $F_{\text{e}}(t_j)(\text{result}_{e_1}(n)) = 1$ and $F_{\text{e}}(t_j)(\text{down}(n)) = 1$. So, for $j = 1, \ldots, n$, define in the context $(p_j, \text{result}_{e}, 0)$ the rule

\[
\text{result}_{e}(0) := (\pi_0 = (\text{result}_{e_1}, \text{down}), \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon;
\]

\[
R_1 = \{1\}, R_0 = (D_e \cup \#)^*.
\]

6. $e = e_1 \subset e_2$: define $A_e = A_{e_1} \cup A_{e_2} \cup \{\text{up}, \text{result}_{e}\}$ and $D_e = D_{e_1} \cup D_{e_2}$. Each region name has the inherited attribute up such that for a region instance $I$ and a region $r$, $F_{\text{e}}(t_I)(\text{up}(n_r)) = 1$ if there exists a region $s$ such that $r \subset s$ and $s \in \llbracket e_2 \rrbracket$. So, for $j = 1, \ldots, n$, define in the context $(p_0, \text{up}, j)$ the rule

\[
\text{up}(j) := (\pi_0 = \varepsilon, \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; R_1 = \emptyset, R_0 = (D_e \cup \#)^*).
\]

For $i = 1, \ldots, n$ and $j = 1, \ldots, n$, define in the context $(p_i, \text{up}, j)$ the rule

\[
\text{up}(j) := (\pi_0 = (\text{up}, \text{result}_{e_2}), \pi_1 = \varepsilon, \ldots, \pi_{n+k} = \varepsilon;
\]

\[
R_1 = \{0\#, 1\#, 11\#\}, R_0 = (D_e \cup \#)^* - R_1.
\]

The rules for $\text{result}_{e}$ are the same as in the previous case, with up instead of down.

7. $e = e_1 < e_2$: define $A_e = A_{e_1} \cup A_{e_2} \cup \{\text{right}, \text{down}, \text{result}_{e}\}$ and $D_e = D_{e_1} \cup D_{e_2}$. The semantic rules for down are the same as in Case 5. Each non-terminal has the inherited attribute right such that for a region instance $I$ and a region $r$, $F_{\text{e}}(t_I)(\text{right}(n_r)) = 1$ if there exists a region $s$ such that $r < s$ and $s \in \llbracket e_2 \rrbracket$. Thus, for $j = 1, \ldots, n$, define in the context $(p_0, \text{right}, j)$ the rule

\[
\text{right}(j) := (\pi_0 = \varepsilon, \pi_1 = (\text{result}_{e_2}, \text{down}), \ldots, \pi_n = (\text{result}_{e_2}, \text{down}),
\]

\[
\pi_{n+1} = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; R_1, R_0 = (D_e \cup \#)^* - R_1,
\]

where $R_1$ is the regular language that contains a string $w_1#aw_2$, with $w_1, w_2 \in \{0, 1\}^*$ and $a \in \{0, 1\}$, if $w_2$ contains a 1. For $i = 1, \ldots, n$ and $j = 1, \ldots, n$, define in the context $(p_i, \text{right}, j)$ the rule

\[
\text{right}(j) := (\pi_0 = \text{right}, \pi_1 = (\text{result}_{e_2}, \text{down}), \ldots, \pi_n = (\text{result}_{e_2}, \text{down}),
\]

\[
\pi_{n+1} = \varepsilon, \ldots, \pi_{n+k} = \varepsilon; R_1, R_0 = (D_e \cup \#)^* - R_1,
\]

where $R_1$ is the regular language that contains a string $aw_1#bw_2$, with $w_1, w_2 \in \{0, 1\}^*$ and $a, b \in \{0, 1\}$, if $w_2$ contains a 1 or $a = 1$. The rules for $\text{result}_{e}$ are the same as in Case 5, with right instead of down.

8. $e = e_1 > e_2$: similar as before.

We need the following definition to state the main result of this section.

**Definition 7.7.** A Region Algebra expression $e$ over $\mathcal{I}$ is empty if for every hierarchical instance $I$ over $\mathcal{I}$, $\llbracket e \rrbracket = \emptyset$. Two Region Algebra expressions $e_1$ and $e_2$ over $\mathcal{I}$ are equivalent if for every hierarchical instance $I$ over $\mathcal{I}$, $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$. 
Theorem 7.8. Testing non-emptiness and equivalence of Region Algebra expressions is in EXPTIME.

Proof. Although every hierarchical instance of \( I = (S_1, \ldots, S_n, \Sigma) \) can be represented as a derivation tree of \( G_I \), not every derivation tree of \( G_I \) is an hierarchical instance. Indeed, if an internal node has no siblings then it represents the same region as its parent. For example, the instance corresponding to the derivation tree \( U(\text{Proc} \circ \text{Func}(a)) \) is not hierarchical because \( \text{Proc} \) and \( \text{Func} \) represent the same region. Also, if a non-terminal node has no children then it represents the empty region, which is not allowed. An extended AG can easily check these conditions by making one bottom–up pass through the tree. Another top–down pass then informs all nodes in the tree whether the tree represents an hierarchical instance.

If \( e \) is a Region Algebra expression, then we define \( \mathcal{F}(e) \) as the extended AG \( \mathcal{F} \), given by Lemma 7.6, that first checks whether the input tree is an hierarchical instance and if so, simulates \( e \); otherwise it assigns false to the result attribute of any node. Hence, \( \mathcal{F}(e) \) is empty if and only if \( e \) is empty. Further, if \( e_1 \) and \( e_2 \) are Region Algebra expressions, then, obviously, \( \mathcal{F}(e_1) \) and \( \mathcal{F}(e_2) \) are equivalent if and only if \( e_1 \) and \( e_2 \) are equivalent. Hence, the result follows by Theorems 6.3 and 6.4. The formal construction is omitted.

8. Expressiveness of extended AGs

We characterize the expressiveness of extended AGs as the queries definable in monadic second-order logic. Monadic second-order logic (MSO) allows the use of set variables ranging over sets of nodes of a tree, in addition to the individual variables ranging over the nodes themselves as provided by first-order logic. We will assume some familiarity with this logic and refer the unfamiliar reader to the book of Ebbinghaus and Flum [20] or the chapter by Thomas [52].

A derivation tree \( t \) can be viewed naturally as a finite relational structure (in the sense of mathematical logic [20]) over the binary relation symbols \( \{E, <\} \) and the unary relation symbols \( \{O_a \mid a \in N \cup T\} \). The domain of \( t \), viewed as a structure, equals the set of nodes of \( t \). The relation \( E \) in \( t \) equals the set of pairs \( (\mathbf{n}, \mathbf{n}') \) such that \( \mathbf{n}' \) is a child of \( \mathbf{n} \) in \( t \). The relation \( < \) in \( t \) equals the set of pairs \( (\mathbf{n}, \mathbf{n}') \) such that \( \mathbf{n}' \neq \mathbf{n}, \mathbf{n}' \) and \( \mathbf{n} \) are children of the same parent and \( \mathbf{n}' \) is a child occurring after \( \mathbf{n} \). The set \( O_a \) in \( t \) equals the set of \( a \)-labeled nodes of \( t \).

MSO can be used in the standard way to define queries. If \( \varphi(x) \) is an MSO-formula, then \( \varphi \) defines the query \( \mathcal{Q} \) defined by \( \mathcal{Q}(t) := \{ \mathbf{n} \in \text{Nodes}(t) \mid t \vDash \varphi[\mathbf{n}]. \}

We start with the easy direction.

Lemma 8.1. Every query expressible by an extended AG is definable in MSO.

Proof. The semantics of an extended AG \( \mathcal{F} \) can readily be defined in MSO. For ease of exposition we assume that all grammar symbols have all attributes, i.e., for every \( X \in N \cup T \), \( \text{Inh}(X) \cup \text{Syn}(X) = A \). We use set variables \( Z_a \), with \( a \) a function from \( A \) to \( D \), to represent assignments of values to attributes. It is not difficult to construct an MSO formula \( \psi((Z_a)_{a \in A \rightarrow D}) \) such that whenever \( t \vDash \psi((Z_a)_{a \in A \rightarrow D}) \) then

- the sets \( (Z_a)_{a \in A \rightarrow D} \) are pairwise disjoint,
- \( \bigcup_a Z_a = \text{Nodes}(t) \), and
- the valuation \( v \) defined as, \( v(a(\mathbf{n})) = a(\mathbf{n}) \) with \( \mathbf{n} \in Z_a \), satisfies \( \mathcal{F} \).
Satisfaction of $\mathcal{F}$ by $v$ is defined as in the proof of Theorem 6.3. The formula $\psi$ just verifies the semantic rules of $\mathcal{F}$. Basically, the latter reduces to the verification of regular languages as is done in the proof of Theorem 6.3; further, regular string languages are easily expressed in MSO [14,52]. The following formula $\psi(x)$ then defines the query expressed by $\mathcal{F}$:

$$\exists Z(x) \in A \rightarrow D \left( \psi((Z(x)) \in A \rightarrow D) \land \bigvee \{ Z(x) \mid \alpha(\text{result}) = 1 \} \right).$$

It guesses a total valuation, verifies that it satisfies $\mathcal{F}$, and selects those nodes for which the result attribute is true. We omit the formal construction of $\psi$ which is straightforward but tedious.

To prove the other direction, we show that extended AGs can compute the MSO-equivalence type of each node of the input tree. Thereto, we introduce some terminology. For a node $n$ of the tree $t$, we write $(t, n)$ to denote the finite structure $t$ expanded with $n$ as a distinguished constant. Let $t_1$ and $t_2$ be two trees, $n_1$ a node of $t_1$, $n_2$ a node of $t_2$ and $k$ a natural number. We write $(t_1, n_1) \equiv_k (t_2, n_2)$ and say that $(t_1, n_1)$ and $(t_2, n_2)$ are $\equiv_k$-equivalent, if for each MSO sentence $\phi$ of quantifier depth at most $k$,

$$(t_1, n_1) \models \phi \iff (t_2, n_2) \models \phi,$$

i.e., $(t_1, n_1)$ and $(t_2, n_2)$ cannot be distinguished by MSO sentences of quantifier depth at most $k$. It follows from the definition that $\equiv_k$ is an equivalence relation. Moreover, $\equiv_k$-equivalence can be nicely characterized by Ehrenfeucht games. The $k$-round MSO game on two structures $(t_1, n_1)$ and $(t_2, n_2)$, denoted by $G_k(t_1, n_1; t_2, n_2)$, is played by two players, the spoiler and the duplicator, in the following way. In each of the $k$ rounds the spoiler decides whether he makes a point move or a set move. When the $i$th move is a point move, he selects one element $p_i \in \text{Nodes}(t_1)$ or $q_i \in \text{Nodes}(t_2)$ and the duplicator answers by selecting one element of the other structure. When the $i$th move is a set move, the spoiler chooses a set $P_i \subseteq \text{Nodes}(t_1)$ or $Q_i \subseteq \text{Nodes}(t_2)$ and the duplicator chooses a set in the other structure. After $k$ rounds there are elements $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$ that were chosen in the point moves in Nodes($t_1$) and Nodes($t_2$), respectively, and there are sets $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ that were chosen in the set moves in Nodes($t_1$) and Nodes($t_2$), respectively. Note that $k = n + l$. The duplicator wins this play if the mapping which maps $p_i$ to $q_i$ (and $n_1$ to $n_2$) is a partial isomorphism from $(t_1, n_1, P_1, \ldots, P_n)$ to $(t_2, n_2, Q_1, \ldots, Q_n)$. That is, for all $i$ and $j$, $p_i \in P_j$ iff $q_i \in Q_j$, and for every atomic formula $\phi(x)$, with $x = x_1, \ldots, x_l$, $(t_1, n_1) \models \phi[p]$ iff $(t_2, n_2) \models \phi[q]$.

We say that the duplicator has a winning strategy in $G_k(t_1, n_1; t_2, n_2)$, or shortly that he wins $G_k(t_1, n_1; t_2, n_2)$, if he can win each play no matter which choices the spoiler makes.

See, e.g., Section 3.1 of the book by Ebbinghaus and Flum [20] for a proof of the next proposition.

**Proposition 8.2.** The duplicator wins $G_k(t_1, n_1; t_2, n_2)$ if and only if

$$(t_1, n_1) \equiv_k (t_2, n_2).$$

The relation $\equiv_k$ has only a finite number of equivalence classes (see, e.g., [20]). We denote the set of these classes by $\Phi_k$. We call the elements of $\Phi_k$ $\equiv_k$-equivalence types (or just $\equiv_k$-types). We denote by $\tau_k(t, n)$ the $\equiv_k$-type of a tree $t$ with a distinguished node $n$; thus, $\tau_k(t, n)$ is the equivalence class of $(t, n)$ w.r.t. $\equiv_k$. It is often useful to think of $\tau_k(t, n)$ as the set of MSO sentences
of quantifier depth at most \( k \) that hold in \((t, n)\). We abuse notation and sometimes write \( \tau^\text{MSO}_k(t, \text{root}(t)) \) for \( \tau^\text{MSO}_k(t) \).

The next proposition contains the main ingredients of the proof of Theorem 8.4. Let \( \varphi(x) \) be an MSO formula of quantifier depth at most \( k \). The first item of the next proposition says that \( t \models \varphi[n] \) only depends on the \( \equiv^\text{MSO}_k \)-type of the subtree rooted at \( n \), i.e., \( \tau^\text{MSO}_k(t, n) \), and on the \( \equiv^\text{MSO}_k \)-type of the envelope of \( t \) at \( n \), i.e., \( \tau^\text{MSO}_k(\overline{t}_n, n) \). Hence, our original problem reduces to the computation of \( \tau^\text{MSO}_k(t, n) \) and \( \tau^\text{MSO}_k(\overline{t}_n, n) \) for each \( n \). The second item (essentially) tells us that \( \tau^\text{MSO}_k(t, n) \) can be computed in a bottom–up manner and from left to right within the siblings of each node. Finally, it follows (essentially) from the third item that \( \tau^\text{MSO}_k(\overline{t}_n, n) \) can be computed in a top–down fashion once the \( \equiv^\text{MSO}_k \)-types of all the \((\overline{t}_m, m)\) are known. The above two pass strategy, first compute the types of all subtrees and then compute the types of all envelopes, forms the core of the proof of Theorem 8.4. The proof of the next proposition is a variation of the well-known composition method, see e.g. [20].

**Proposition 8.3.** Let \( k \) be a natural number, \( \sigma \) be a label, \( t \) and \( s \) be two trees, \( n \) be a node of \( t \) with children \( n_1, \ldots, n_n \), and \( m \) be a node of \( s \) with children \( m_1, \ldots, m_m \). Let the label of \( n \) and \( m \) be \( \sigma \).

1. If \((\overline{t}_n, n) \equiv^\text{MSO}_k (\overline{s}_m, m) \) and \((t_n, n) \equiv^\text{MSO}_k (s_m, m) \), then \((t, n) \equiv^\text{MSO}_k (s, m) \).

2. If \((\sigma(t_{n_1}, \ldots, t_{n_{n-1}}), \text{root}) \equiv^\text{MSO}_k (\sigma(s_{m_1}, \ldots, s_{m_{m-1}}), \text{root}) \) and \((t_n, n) \equiv^\text{MSO}_k (s_m, m) \), then \((t, n) \equiv^\text{MSO}_k (s, m) \).

3. For \( i \in \{1, \ldots, n\} \) and \( j \in \{1, \ldots, m\} \), if

   - \((\overline{t}_n, n) \equiv^\text{MSO}_k (\overline{s}_m, m) \),

   - \((\sigma(t_{n_1}, \ldots, t_{n_{i-1}}), \text{root}) \equiv^\text{MSO}_k (\sigma(s_{m_1}, \ldots, s_{m_{j-1}}), \text{root}) \),

   - \((\sigma(t_{n_{i+1}}, \ldots, t_{n_n}), \text{root}) \equiv^\text{MSO}_k (\sigma(s_{m_{j+1}}, \ldots, s_{m_m}), \text{root}) \), and

   - the label of \( n_i \) equals the label of \( m_j \),

   then \((\overline{t}_n, n_i) \equiv^\text{MSO}_k (\overline{s}_m, m_j) \).

**Proof.** Consider the first item. By Proposition 8.2, it suffices to show that the duplicator wins \( G^\text{MSO}_k(t, n; s; m) \). We already know that he wins the subgames \( G^\text{MSO}_k(t_n, s_m; m) \) and \( G^\text{MSO}_k(\overline{t}_n, n; \overline{s}_m; m) \). The duplicator, therefore, combines these winning strategies as follows to obtain a winning strategy in \( G^\text{MSO}_k(t, n; s; m) \). If the spoiler makes a point move then the duplicator answers corresponding to his winning strategy in the relevant subgame. If the spoiler makes a set move in, say, \( t \), choosing the sets \( P_1 \subseteq \text{Nodes}(t_n) \) and \( P_2 \subseteq \text{Nodes}(\overline{t}_n) \), then the duplicator responds with the set \( Q_1 \cup Q_2 \), where \( Q_1 \) is the answer to \( P_1 \) in the subgame \( G^\text{MSO}_k(t_n, s_m; m) \) and \( Q_2 \) is the answer to \( P_2 \) in the subgame \( G^\text{MSO}_k(\overline{t}_n, n; \overline{s}_m; m) \).

Note that this strategy is well defined. Indeed, \((t_n, n) \) and \((\overline{t}_n, n) \) \((s_m, m) \) and \((\overline{s}_m, m) \) have only \( n \) \( m \) in common and due to the fact that \( n \) and \( m \) are distinguished constants in both subgames, the duplicator is forced to pick \( n \) whenever the spoiler picks \( m \), and vice versa. At the end of a game, the selected vertices define partial isomorphisms for the two pairs of respective substructures. As there is no relation in the vocabulary that can relate a node from, say, \( t_n \) to a node from \( \overline{t}_n \) (apart from \( n \), these mappings are also partial isomorphisms for the whole structures. Hence, the above strategy is also winning.

We next focus on the third case and leave the second case to the reader. Here, there are altogether four subgames including the trivial game in which one structure consists only of \( n_i \) and the other of \( m_j \). The winning strategy in the game on \((\overline{t}_n, n_i) \) and \((\overline{s}_m, m_j) \) just combines the winning strategies in those
four subgames (as explained for the first item). Again, at the end of a game, the selected vertices define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the preservation of the relations $<$ and $E$ between the chosen elements, and the distinguished constants $n_i$ and $m_j$. This immediately follows from the following observations. The distinguished constants in the subgames make sure that (a) whenever in the game on $(t_{n_i}, n_i)$ and $(s_{m_j}, m_j)$ a child of $n_i$ (m) is chosen, the duplicator has to reply with a child of $m_j (n)$; and, (b) whenever $n_i$ (m) is chosen, the duplicator has to reply with $m_j (n)$. Additionally, the position of the subtrees in the whole tree make sure that $<$ is preserved w.r.t. $n_i$ and $m_j$.

We are ready to prove the main theorem of this section. Recall that the ECFG is given.

**Theorem 8.4.** A query is expressible by an extended AG if and only if it is definable in MSO.

**Proof.** The only-if direction was already given in Lemma 8.1.

Let $\varphi(x)$ be an MSO formula of quantifier depth $k$. We define an extended AG $\mathcal{F}$ expressing the query defined by $\varphi$. Define $D = \Phi_k \cup \{0, 1\} \cup N \cup T$ and $A = \{\text{env, sub, result, lab}\}$, where env is inherited for all grammar symbols except for the start symbol for which it is synthesized, and sub and result are synthesized for all non-terminals and inherited for all terminals. The intended meaning is the following: for a node $n$ of a tree $t$,

- $\mathcal{F}(t)(\text{sub}(n)) = \tau_k^{\text{MO}}(t_{n_i}, n_i)$,
- $\mathcal{F}(t)(\text{env}(n)) = \tau_k^{\text{MO}}(t_{n_i}, n_i)$, and
- $\mathcal{F}(t)(\text{result}(n)) = 1$ if and only if $t \models \varphi[n]$.

By Proposition 8.3(1), $t \models \varphi[n]$ only depends on $\tau_k^{\text{MO}}(t_{n_i}, n_i)$ and $\tau_k^{\text{MO}}(t_{n_i}, n_i)$. Hence, $\mathcal{F}(t)(\text{result}(n))$ only depends on the attribute values $\mathcal{F}(t)(\text{env}(n))$ and $\mathcal{F}(t)(\text{sub}(n))$.

As already hinted upon above, the extended AG we construct, works in two passes. In the first bottom–up pass all the sub attributes are computed (using the regular languages SUB, defined below); in the subsequent top–down pass all the env attributes are computed (using the regular languages ENV, defined below). Recall our convention that the start symbol cannot appear in the left-hand side of a production. Hence, whenever we encounter a node labeled with the start symbol, we know it is the root and can initiate our top–down pass.\footnote{As the sole purpose of this technical convention is to be able to identify the root of the input tree, it can easily be dispensed with by adding so-called root rules to the attribute grammar formalism (see, e.g., Giegerich for a definition of standard attribute grammars with root rules [23]).} During this second pass, there is enough information at each node $n$ to decide whether $t \models \varphi[n]$.

We next define the regular languages SUB, which we use to compute $\equiv^k_\text{MSO}$-types of subtrees in a bottom–up fashion. We again abbreviate $\tau_k^{\text{MO}}(t, \text{root}(t))$ by $\tau_k^{\text{MO}}(t, \text{root})$. Define for $\theta \in \Phi_k$ and $X \in N$ the language $\text{SUB}(X, \theta)$ over $\Phi_k$ as follows:

\[ \theta_1 \ldots \theta_n \in \text{SUB}(X, \theta) \]

if there exist trees $t_1, \ldots, t_n$ such that for $i = 1, \ldots, n$, $\tau_k^{\text{MO}}(t_i, \text{root}) = \theta_i$ and $\tau_k^{\text{MO}}(X(t_1, \ldots, t_n), \text{root}) = \theta$. We show that $\text{SUB}(X, \theta)$ is a regular language.
Claim 8.5. Let \( X \in N \) and \( \theta \in \Phi_k \). There exists a DFA \( M = (S, \Phi_k, \delta, s_0, F) \) accepting \( \text{SUB}(X, \theta) \).

Proof. Define \( M = (S, \Phi_k, \delta, s_0, F) \) as the DFA where \( S = \Phi_k \cup \{s_0\} \) and \( F = \{\theta\} \). Define the transition function as follows: for all \( \theta', \theta_1, \theta_2 \in \Phi_k \),

- \( \delta(s_0, \theta') := \tau_k^\text{MSO}(X(t), \text{root}) \) with \( \tau_k^\text{MSO}(t, \text{root}) = \theta' \) whenever such a \( t \) exists, and
- \( \delta(\theta_1, \theta_2) := \tau_k^\text{MSO}(t, \text{root}) = \theta_1, \tau_k^\text{MSO}(t, \text{root}) = \theta_2 \).

By Proposition 8.3(2), it does not matter which trees in the equivalence classes \( \theta', \theta_1, \) and \( \theta_2 \) we take. This proves the claim.

To prove the claim, note that for all trees \( t_1, \ldots, t_n \), if \( \tau_k^\text{MSO}(t_i, \text{root}) = \theta_i \), for \( i = 1, \ldots, n \), and \( \tau_k^\text{MSO}(X(t_1, \ldots, t_n), \text{root}) = \theta' \), then \( \delta^*(s_0, \theta_1 \cdots \theta_n) = \theta' \).

Define for \( \theta \in \Phi_k \), the language \( \text{ENV}(\theta) \) over \( \Phi_k \cup \{\#\} \) as follows:

\[
\overline{\theta} = \theta_0 \theta_1 \cdots \theta_{i-1} # \theta_i \theta_{i+1} \cdots \theta_n \in \text{ENV}(\theta)
\]

iff

- \( \theta_j \in \Phi_k \) for \( j = 0, \ldots, n \), and
- there exists a tree \( t \) with a node \( n \) of arity \( n \) (for some \( n \)) such that \( \tau_k^\text{MSO}(\overline{t}, \text{root}) = \theta_0, \tau_k^\text{MSO}(\overline{t}, n) = \theta_1 \), and \( \tau_k^\text{MSO}(t, \text{root}) = \theta_2 \).

By Proposition 8.3(3), \( \overline{\theta} \in \text{ENV}(\theta) \) only depends on \( \tau_k^\text{MSO}(\overline{t}, n) \), \( \tau_k^\text{MSO}(\overline{t}, n_i) \), and \( \tau_k^\text{MSO}(X(t_{n+1}, \ldots, t_n), \text{root}) \), and the label of \( n_i \) which in turn only depends on \( \tau_k^\text{MSO}(t, n_i) \). In terms of the automaton \( M \) of Claim 8.5, \( \overline{\theta} \in \text{ENV}(\theta) \) only depends on \( \theta_0, \delta^*(s_0, \theta_1 \cdots \theta_{i-1}) \), \( \delta^*(s_0, \theta_{i+1} \cdots \theta_n) \), and \( \theta_i \). It is, hence, not difficult to construct an automaton accepting \( \text{ENV}(\theta) \). Indeed, such an automaton stores \( \theta_0 \) in its state; then simulates \( M \) until it reaches the symbol \#, this gives the state \( \delta^*(s_0, \theta_1 \cdots \theta_{i-1}) \); hereafter \( M \) stores \( \theta_i \) in its state and again simulates \( M \) until the end of the string which gives the state \( \delta^*(s_0, \theta_1 \cdots \theta_n) \); \( M \) then accepts if these four types determine the type \( \theta \) as explained above.

By Proposition 8.3, for all \( X \in N \) and \( \theta_1, \theta_2 \in \Phi_k \), if \( \theta_1 \neq \theta_2 \) then \( \text{SUB}(X, \theta_1) \cap \text{SUB}(X, \theta_2) = \emptyset \) and \( \text{ENV}(\theta_1) \cap \text{ENV}(\theta_2) = \emptyset \). Also, for all \( X \in N \), \( \bigcup_{\theta \in \Phi_k} \text{SUB}(X, \theta) = \Phi_k^* \) and \( \bigcup_{\theta \in \Phi_k} \text{ENV}(\theta) = \Phi_k^* \# \Phi_k^* \).

We are finally ready to define the semantic rules of \( \mathcal{F} \). For every production \( X \to r \), define in the context \( (X \to r, \text{sub}, 0) \) the rule

\[
\text{sub}(0) := \langle \pi_0 = \epsilon, \pi_1 = \text{sub}, \ldots, \pi_{|r|} = \text{sub}; R_0 = \text{SUB}(X, \theta) \rangle_{\theta \in \Phi_k}, R_0 = D^* - \Phi_k^*.
\]

Note that \( R_0 \) is irrelevant. The only reason \( R_0 \) is non-empty is the totality requirement on extended AGs (the union of all \( R_d \)’s should be \( D^* \)). As usual, the \( R_d \)’s that are not mentioned are defined as the empty set. For every \( i \) such that \( r(i) = \sigma \) is a terminal define in the context \( (X \to r, \text{sub}, i) \) the rule

\[
\text{sub}(i) := \langle \pi_0 = \epsilon, \pi_1 = \epsilon, \ldots, \pi_{|r|} = \epsilon; R_{\theta_0} = \# \rangle, R_0 = D^* \# D^* - \{\#\}.
\]

The above rule just assigns the type \( \theta_{\sigma} = \tau_k^\text{MSO}(\sigma, \text{root}) \) to every terminal \( \sigma \). For \( i = 1, \ldots, |r| \), define in the context \( (X \to r, \text{env}, i) \) the rule

\[
\text{env}(i) := \langle \pi_0 = \text{env}, \pi_1 = \text{sub}, \ldots, \pi_{|r|} = \text{sub}; R_0 = \text{ENV}(\theta) \rangle_{\theta \in \Phi_k}, R_0 = ((D - \Phi_k) \cup \{\#\})^*.
\]
For the start symbol, define in the context \((U \rightarrow r, \text{env}, 0)\) the rule

\[
\text{env}(0) := \langle \pi_0 = \varepsilon, \pi_1 = \varepsilon, \ldots, \pi_{|r|} = \varepsilon; R_{\theta(U)} = \{\varepsilon\}, R_0 = DD^* \rangle,
\]

where \(\theta(U) = \tau_k^{\text{MSO}}(t(U), \text{root})\). Finally, add in the context \((X \rightarrow r, \text{result}, 0)\) the rule

\[
\text{result}(0) := \langle \pi_0 = (\text{env, sub}), \pi_1 = \varepsilon, \ldots, \pi_{|r|} = \varepsilon; R_1, R_0 = D^* - R_1 \rangle,
\]

and for every \(i\) such that \(r(i)\) is a terminal, add in the context \((X \rightarrow r, \text{result}, i)\) the rule

\[
\text{result}(i) := \langle \pi_0 = \varepsilon, \pi_1 = \varepsilon, \ldots, \pi_{i-1} = \varepsilon, \pi_i = (\text{env, sub}), \\
\pi_{i+1} = \varepsilon, \ldots, \pi_{|r|} = \varepsilon; R_1, R_0 = (D \cup \{\#\})^* - R_1 \rangle,
\]

where \(R_1\) consists of those three letter strings \(#0_10_2 \in #D^2\) for which there exists a tree \(t\) with a node \(n\), with \(\tau_k^{\text{MSO}}(t_n, n) = 0_1, \tau_k^{\text{MSO}}(t_n, \text{root}) = 0_2\), and \(t \models \phi[n]\).

**Remark 8.6.** It remains to argue that the construction in the proof of Theorem 8.4 is effective. First observe that for every \(k\) there are only finitely many pairwise non-equivalent formulas of quantifier depth \(k\) with a fixed number of free variables (see, e.g., [20]). Call this set \(S_k\). Members of \(S_k\) can effectively be constructed by enumerating formulas in prenex-normal form with their body in disjunctive normal form and only keeping those that are pairwise equivalent. The latter is decidable as MSO over unranked trees is [41]. The effectiveness of the construction then follows from the fact that \(\theta\) can both be represented by a member \((t_0, n_0)\) of the corresponding \(\equiv_k^{\text{MSO}}\) class and the conjunction of formulas that hold in \((t_0, n_0)\). For every \(\equiv_k^{\text{MSO}}\) class \(\theta\), we can construct a tree \(t_0\) with node \(n_0\) such that \(\tau_k^{\text{MSO}}(t_0, n_0) = \theta\) by constructing a tree automaton for this equivalence class and then constructing a tree which the automaton accepts.

9. Discussion

In other work [41], Schwentick and the present author defined query automata to query structured documents. Query automata are two-way automata over (un)ranked trees that can select nodes depending on the current state and on the label at these nodes. Query automata can express precisely the unary MSO definable queries and have an EXPTIME-complete equivalence problem. This makes them look rather similar to extended AGs. The two formalisms are, however, very different in nature. Indeed, query automata constitute a procedural formalism that has only local memory (in the state of the automaton), but which can visit each node more than a constant number of times. Attribute grammars, on the other hand, are a declarative formalism, whose evaluation visits each node of the input tree only a constant number of times (once for each attribute). In addition, they have a distributed memory (in the attributes at each node). It is precisely this distributed memory which makes extended AGs particularly well-suited for an efficient simulation of Region Algebra expressions. It is, therefore, not clear whether there exists an efficient translation from Region Algebra expressions into query automata.

Extended AGs can only express queries that retrieve subtrees from a document. It would be interesting to see whether the present formalism can be extended to also take restructuring of documents into account. A related paper in this respect is that of Crescenzi and Mecca [18]. They define an interesting formalism for the definition of wrappers that map derivation trees of regular grammars to relational databases. Their
formalism, however, is only defined for regular grammars and the correspondence between actions (i.e., semantic rules) and grammar symbols occurring in regular expressions is not so flexible as for extended AGs. Other work that uses attribute grammars in the context of databases includes work of Abiteboul et al. [2] and Kilpeläinen et al. [32].

Acknowledgments

This paper is inspired by the work of Brüggemann-Klein, Murata and Wood. I thank them for providing me with a first draft of [13]. I thank Thomas Schwentick, Jan Van den Bussche, and Dirk Van Gucht for helpful discussions on topics treated in this paper. I especially thank one of the anonymous referees whose detailed comments tremendously improved the presentation of the paper.

References


