UMEÅ UNIVERSITY

Order-Preserving Graph Grammars

Petter Ericson

DOCTORAL THESIS, FEBRUARY 2019
DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SWEDEN

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

pettter@cs.umu.se

ISBN 978-91-7855-017-3
ISSN 0348-0542
UMINF 19.01

Front cover by Petter Ericson
Printed by UmU Print Service, Umeå University, 2019.

It is good to have an end to journey toward;
but it is the journey that matters, in the end.

URSULA K. LE GUIN

Abstract

The field of *semantic modelling* concerns formal models for semantics, that is, formal structures for the computational and algorithmic processing of *meaning*. This thesis concerns formal *graph languages* motivated by this field. In particular, we investigate two formalisms: Order-Preserving DAG Grammars (OPDG) and Order-Preserving Hyperedge Replacement Grammars (OPHG), where OPHG generalise OPDG.

Graph parsing is the practise of, given a graph grammar and a graph, to determine if, and in which way, the grammar could have generated the graph. If the grammar is considered fixed, it is the *non-uniform* graph parsing problem, while if the grammars is considered part of the input, it is named the *uniform* graph parsing problem. Most graph grammars have parsing problems known to be NP-complete, or even exponential, even in the non-uniform case. We show both OPDG and OPHG to have polynomial uniform parsing problems, under certain assumptions.

We also show these parsing algorithms to be suitable, not just for determining membership in graph languages, but for computing weights of graphs in *graph series*.

Additionally, OPDG is shown to have several properties common to regular languages, such as MSO definability and MAT learnability. We moreover show a direct correspondence between OPDG and the regular tree grammars.

Finally, we present some limited practical experiments showing that real-world semantic graphs appear to mostly conform to the requirements set by OPDG, after minimal, reversible processing.

Populärvetenskaplig sammanfattning

Inom språkvetenskap och datalingvistik handlar mycket forskning om att på olika sätt analysera *strukturer* inom språk; dels syntaktiska strukturer – vilka ordklasser som kommer före andra och hur de i satser och satsdelar kan kombineras för korrekt meningsbyggnad, och dels hur olika ords *mening* eller *semantik* kan relatera till varandra och till idéer, koncept, tankar och ting. Denna avhandling behandlar formella datavetenskapliga modeller för just sådan *semantisk modellering*. I vårt fall representeras vad en mening betyder som en *graf*, bestående av noder och kanter mellan noder, och de formella modeller som diskuteras är *grafgrammatiker*, som avgör vilka grafer som är korrekta och inte.

Att med hjälp av en grafgrammatik avgöra om och hur en viss graf är korrekt kallas för *grafparsing*, och är ett problem som generellt är beräkningsmässigt svårt – en grafparsingalgoritm tar i många fall exponentiell tid i grafens storlek att genomföra, och ofta påverkas dessutom körtiden av grammatikens sammansättning och storlek.

I den här avhandlingen beskrivs två relaterade modeller för semantiskt modellering – Ordningsbevarande DAG-grammatiker (OPDG) och Ordningsbevarande Hyperkantsomskrivningsgrammatiker (OPHG). Vi visar att grafparsingproblemet för OPDG och OPHG är effektivt lösbart, och utforskar vad som behöver gälla för att en viss grammatik skall vara en OPHG eller OPDG.

Acknowledgements

There are very few books that can be attributed solely to a single person, and this thesis is no exception to that rule. Listing all of the contributors is an exercise in futility, and thus my lie in the acknowledgement section of my licentiate thesis is exposed – this will by necessity if not by ambition be an incomplete selection of people who have helped me get to this point.

First off, my advisor Henrik Björklund and co-advisor Frank Drewes both deserve an enormous part of the credit that is due from this thesis. They have discussed, co-authored, taught, coaxed, calmed down, socialised, suggested, edited, and corrected, all as warranted by my ramblings and wanderings in the field and elsewhere, and all of that has acted to make me a better researcher and a better person both. Thank you.

Second, Linn, my better half, has helped with talk, food, hugs, kisses, walks, travels, schedules, lists (so many lists) and all sort of other things that has not only kept me (more or less) sane, but in any sort of state to be capable of finishing this thesis. There is no doubt in my mind that it would not have been possible, had I not had you by my side. You are simply the best.

Third, the best friend I could never deserve and always be thankful for, Philip, who has simply been there through it all, be it on stage, at the gaming table, or in the struggles of academia. Your help with this thesis made it at least $134(\pm 3)\%$ better in every single respect, and hauled me back from the brink, quite literally. You are *also*, somehow, the best.

I have also had the pleasure and privilege to be part of an awesome group of people known as the Foundations of Language Processing research group, which have at different points during my PhD studies contained people such as my co-authors Johanna and Florian, my bandmate Niklas, my PhD senior Martin, and juniors Anna, Adam and Yonas. Suna and Mike are neither co-authors nor common PhD student with me, but have been magnificient colleagues and inspirations nonetheless. I have enjoyed our friday lunches, seminars and discussion greatly, and will certainly try to set up something similar wherever I end up.

Many other colleagues have helped me stay on target in various ways, such as my partner in course crime Jan-Erik, the compulsive crosswords-solvers including Tomas, Carina, Helena, Niklas, Niclas, Helena, Lars, Johan, Mattias, Pedher, and many more coworkers of both the crossword-solving and -avoiding variety.

My family has also contributed massively with both help and inspiration during my PhD studies. My parents Curry and Lars both supplying rides, talks, jams, academic insights, dinners, berries, mushrooms, practically infinite patience, and much more.

My sister Tove and her husband Erik providing a proof-of-concept of Dr. Ericson, showing that it can indeed be done, and that it can be done while being and remaining some of the best people I know, and all this while my niece Isa and nephew Malte came along as well.

Too many friends and associations have brightened my days and evenings during these last years to list them all, but I will make the attempt. In no particular order, thanks to Olow, Diana, Renhornen, JAMBB, Lochlan, Eric, Jenni, Bengt, Birgit, Kelly, Tom, MusicTechFest, Björn, Linda, Erik, Filip, Hanna (who got me into this whole thing), Hanna (who didn't), Michela, Dubber (we'll get that paper written soon), Sofie, Alexander, Maja, Jonathan, Peter, Niklas, Oscar, Hanna (a recurring pattern), Mats, Nick, Viktor, Anna, Ewa, Snösvänget, Avstamp, Isak, Samuel, Kristina, Ivar, Helena, Christina, Britt, Louise, Calle, Berit, Thomas, Kerstin, Lottis, Ola, Mattias, Mikael, Mikael, Tomas, Mika, Benjamin, and the rest of the Umeå Hackerspace gang, Offer, André, Four Day Weekend, LESS, Mikael, Staffan, Lisa, Jesper, Henrik, John, Mats, Carlos, Valdemar, Axel, Arvid, Christoffer, Tomas, Becky, Jocke, Jennifer, Jacob, Jesper, Ellinor, Magne, Hanna (yes really), Johan, Lennart, LLEO, Jet Cassette, Calzone 70, Veronica, Johan, Container City, Malin, Sanna, Fredrik, Maja, Mats, and everyone that have slipped my mind for the moment and will remember seconds after sending this to print. Thank you all!

Preface

The following papers make up this Doctoral Thesis, together with an introduction.

Paper I — Henrik Björklund, Frank Drewes, and Petter Ericson.
Between a Rock and a Hard Place – Parsing for Hyperedge Replacement DAG Grammars.
In *10th International Conference on Language and Automata Theory and Applications (LATA 2016), Prague, Czech Republic*, pp. 521-532, Springer, 2016.

Paper II — Henrik Björklund, Johanna Björklund, and Petter Ericson.
On the Regularity and Learnability of Ordered DAG Languages.
In Arnaus Carayol and Cyril Nicaud, editors, *22nd International Conference on the Implementation and Application of Automata (CIAA 2017), Marne-la-Vallée, France,* volume 10329 of *Lecture Notes in Computer Science*, pp. 27-39, Springer 2017.

Paper III — Henrik Björklund, Johanna Björklund, and Petter Ericson.
Minimisation and Characterization of Order-preserving DAG Grammars.
Technical Report UMINF 18.15 Dept. Computing Sci., Umeå University, `http://www8.cs.umu.se/research/uminf/index.cgi`, 2018. Submitted

Paper IV — Henrik Björklund, Frank Drewes, and Petter Ericson.
Uniform Parsing for Hyperedge Replacement Grammars.
Technical Report UMINF 18.13 Dept. Computing Sci., Umeå University, `http://www8.cs.umu.se/research/uminf/index.cgi`, 2018. Submitted

Paper V — Henrik Björklund, Frank Drewes, and Petter Ericson.
Parsing Weighted Order-Preserving Hyperedge Replacement Grammars.
Technical Report UMINF 18.16 Dept. Computing Sci., Umeå University, `http://www8.cs.umu.se/research/uminf/index.cgi`, 2018.

Additionally, the following technical report and paper were completed during the course of the PhD program.

Paper I Petter Ericson.
A Bottom-Up Automaton for Tree Adjoining Languages.
Technical Report UMINF 15.14 Dept. Computing Sci., Umeå University, `http://www8.cs.umu.se/research/uminf/index.cgi`, 2015.

Paper II Henrik Björklund and Petter Ericson.
A Note on the Complexity of Deterministic Tree-Walking Transducers.
In *Fifth Workshop on Non-Classical Models of Automata and Applications (NCMA)*, pp. 69-84, Austrian Computer Society, 2013.

Contents

CHAPTER 1
Introduction

This thesis concerns the study of graphs, and grammars and automata working on graphs, with applications in the processing of human language as well as other fields. A new formalism with two variants is presented and various desirable features are shown to hold, such as efficient (weighted) parsing, and for the limited variant, MSO definability, MAT learnability, and a normal form.

It also presents novel, though limited, practical work using these formalisms, and aims to show their usefulness in order to motivate further study.

1.1 The Study of Languages

The *study of languages* has at minimum two very distinct meanings. This thesis concerns both.

The first, and probably most intuitive, sense of "study of language" concerns *human languages*, which is the kind of languages we humans use to communicate ideas, thoughts, concepts and feelings. The study of such languages is an immense field of research, including the whole field of linguistics, but also parts of informatics, musicology, various fields studying *particular* languages such as English or Swedish, philosophy of language, and many others. It concerns what human language is, how languages work, and how they can be applied. It also concerns how understanding of human language and various human language tasks such as translation and transcription can be formalised or computerised, which is where the work presented in this thesis intersects the subject.

Though many of the applications mentioned in this thesis make reference to *natural language processing* (NLP), it may be appropriate to mention that many of the techniques are in fact also potentially useful for processing *constructed languages* such as Klingon,[1] Quenya,[2] Esperanto,[3], Lojban[4] and toki pona[5]. The language of music also shares many of the same features, and musical applications are used later in this introduction to illustrate a practical use case of the results.

[1] From Gene Roddenberry's "Star Trek"
[2] From J.R.R. Tolkien's "The Lord of the Rings"
[3] An attempt at a "universal" language, with a grammar without exceptions and irregularities.
[4] A "logical" language where the grammar is constructed to minimise ambiguities.
[5] A "Taoist" language, where words are generally composites of a "minimal" amount of basic concepts.

The particular field that has motivated the work presented in this thesis is that of *semantic modelling*, which seeks to capture the semantics, or *meaning* of various language objects, e.g. sentences, in a form suitable for further computational and algorithmic processing. In short, we wish to move from representing some real-world or imaginary concept using *natural* language to representing the same thing using more *formal* language.

This bring us to the second meaning of "language" and its study. For this we require some mathematical and theoretical computer science background. In short, given a (usually infinite) universe (set) of objects, a (formal) *language* is any subset of the universe (including the empty set and the complete universe). The study of these languages, their definitions, applications, and various other properties, is, loosely specified, the field of *formal language theory*. Granted, the given definition is *very* wide, bordering on unusable, and we will spend Chapters 3 to 5 defining more specifically the field and subfield that is the topic of this thesis.

1.2 Organisation

The organisation of the rest of this thesis proceeds as follows: First, we present the field of semantic modelling through a brief history of the field, after which we introduce formal language theory in general, including the relevant theory of finite automata, logic, and various other fields. We then discuss the formalisms that form the major contributions in the papers included in this thesis. We briefly introduce a number of related formalisms, before turning to the areas and applications which we aim to produce practical results for, and the minor practical results themselves. We conclude the introduction with a section on future work, both for developing and using our formalisms. Finally, the five papers that comprise the major scientific contributions in this thesis are included.

CHAPTER 2
Semantic modelling

The whole field of human language is much too vast to give even a cursory introduction to in a reasonable way, so let us focus on the specific area of study that has motivated the theoretical work presented here.

From the study of human language in its entirety, let us first focus on the area of *natural language processing*, which can be very loosely described as the study of how to use computers and algorithms for natural language tasks such as translation, transcription, natural language interfaces, and various kinds and applications of natural language understanding. This, while being more specific than the whole field of human language, is thus still quite general.

The specific area within natural language processing that interests us is *semantic processing*, and even more specifically, *semantic modelling*. That is, we are more interested in the structure of *meaning*, than that of *syntax*. Again, this is a quite wide field, which has a long and varied history, but the problem can be briefly stated as follows: How can we represent the meaning of a sentence in a way that is both reasonable and useful?

2.1 History

Arguably, this field has predecessors all the way back to the beginning of the Enlightenment with the attempts of Francis Lodwick, John Wilkins and Gottfried Wilhelm Leibniz among others to develop a "philosophical language" which would be able to accurately represent facts of the world without the messy abstractions, hidden contexts and ambiguities of natural language. This was to be accomplished in Wilkins conception [Wil68] through, on the one hand, a "scientifically" conceived writing system based on the anatomy of speech, on the other hand, an unambiguous and regular syntax with which to construct words and sentences, and on the gripping hand,[1] a well-known and unambiguous way to refer to things and concepts, and their relations. Needless to say these attempts, though laudable and resulting in great insights, ended with no such language in use, and the worlds of both reality and imagination had proven to be much more complex and fluid than the strict categories and forty

[1] This is a somewhat oblique reference to the science fiction novel "A Mote in Gods Eye" by Larry Niven and Jerry Pournelle. The confused reader may interpret this as "on the third hand, and most importantly".

all-encompassing groupings (or genera) of Wilkins.

Even though the intervening years cover many more interesting and profound insights, the next major step that is relevant to this thesis occurs in the mid-twentieth century, with the very first steps into implementing general AI on computers. Here, various approaches to *knowledge representation* could be tested "in the field", by attempting to build AI systems using them. Initially, projects like the General Problem Solver [NSS59] tried to, once again, deal with the world and realm of imagination in a complete, systematic way. However, the complexities and ambiguities of the real world once again gradually asserted themselves to dissuade this approach. Instead, *expert systems* became the norm, where the domain was very limited, e.g. to reasoning about medical diagnoses. Here, the meaning of sentences could be reduced to simple logical formulae and assertions, which could then be processed to give a correct output given the inputs.

In parallel, the field of natural language processing was in its nascent stages, with automated translation being widely predicted to be an easy first step to fully natural language interfaces being standard for all computers. This, too, was quickly proven to be an overly optimistic estimation of the speed of AI research, but led to an influx of funding, kick-starting the field and its companion field of *computational linguistics*. Initial translator systems dispensed with any pretence at semantic understanding or representation, and used pure lexical word for word correspondences between languages to translate text from one language to another, with some special rules in place for certain reorderings, deletions and insertions. Many later attempts were built on similar principles, but sometimes used an intermediate representation, or *interlingua*, as a step between languages. This can be seen as a kind of semantic representation.

After several decades of ever more complex rules and hand-written grammars, the real world once again showed itself to be much too complex to write down in an exact way. Meanwhile, an enormous amount of data had started to accumulate in ever more computer accessible formats, and simple statistical models trained on such data started seeing success in various NLP tasks. Recent examples of descendants of such models include all sorts of neural network and "deep learning" approaches.

With the introduction of statistical models, we have essentially arrived at a reasonable picture of the present day, though the balance between data-driven and hand-written continues to be a difficult one, as is the balance between complexity and expressivity of the chosen models. An additional balance that has recently come to the fore is the balance between efficiency and *explainability* – that is, if we construct a (usually heavily data-driven) model and use it for some task, how easy is it so see *why* the model behaves as it does, and gives the results we obtain? All of these separate dimensions of course interact, sometimes in unexpected ways.

2.2 Issues in Semantic Modelling

With this history in mind, let us turn to the practicalities of representing semantics. The two keywords in our question above is "reasonable" and "useful" – they require some level of judgement as to for whom and what purpose the representation should

be reasonable and useful. As such, this question has much in common with other questions in NLP research. Should we focus on making translations that seem good to professional translators and bilinguals, or should we focus on making the translations explicit, showing the influence of each part of the input to the relevant parts of the output? Is it more important to use data structures that mimic what "really" is going on in the mind, or should we use abstractions that are easier to reason about, even though they may be faulty or inflexible, or should we just use "whatever works best", for some useful metric?

Thus the particular semantic representation chosen depends very much on the research question and on subjective valuations. In many instances, the semantics are less important than the function, and thus a relatively opaque representation may be chosen, such as a simple vector of numbers based on word proximity (word embeddings). In others, the computability and manipulation of semantics is the central interest, and thus the choice is made to represent semantics using logical structures and formulae.

In this thesis, we have chosen to explore formalisms for semantic *graphs*, that represent the meaning of sentences using connections between concepts. However, to properly define and discuss these requires a bit more background of a more formal nature, which will be provided in the following chapters.

CHAPTER 3

String languages

With some background established in the general field of natural language processing, let us turn to theory.

To define *strings*, we first need to define *alphabets*: these are (usually finite) sets of distinguishable objects, which we call *symbols*. A *string over the alphabet* Σ is any sequence of symbols from the alphabet, and a *string language* (over the same) is any set of such strings. A set of languages is called a *class* of languages.

Thus the string *"aababb"* is a string over the alphabet $\{a,b\}$ (or any alphabet containing *a* and *b*), while the string *"the quick brown fox jumps over the lazy dog"* is a string over the alphabet of lowercase letters *a* to *z* and a space character (or any superset thereof).

Further, any finite set of strings, such as {"*a*","*aa*","*aba*"} is a string language, but we can also define infinite string languages such as "all strings over the alphabet $\{a,b\}$ containing an even number of *a*'s", or "any string over the English alphabet (plus a space character) containing the word *supercalifragilisticexpialidocious*".

For classes of languages, we can again look at finite sets of languages, though it is generally more interesting to study infinite sets of infinite languages. Let us look closer at such a class – the well-studied class of *regular string languages* (REG). We define this class first inductively using *regular expressions*, which were first described by Stephen Kleene in [Kle51]. In the following, *e* and *f* refer to regular expressions, while *a* is a symbol from the alphabet.

- A symbol *a* is a regular expression defining the language "a string containing only the symbol *a*".
- A concatenation $e \cdot f$ defines the language "a string consisting of first a string from the language of *e*, followed by a string from the language of *f*". We often omit the dot, leaving ef.
- An alternation (or union) $(e)|(f)$ defines the language "either a string from the language of *e*, or one from the language of *f*"
- A repetition e^* defines the language "zero or more concatenated strings from the language of *e*" [1]

[1] This is generally called a Kleene star, from Stephen Kleene.

Thus, the regular expression $(a)|(aa)|(aba)$ defines the finite string language $\{$"a","aa","aba"$\}$, while $(b^*ab^*a)^*b^*$ is one way of writing the language "all strings over $\{ab\}$ with an even number of a's".

3.1 Automata

Let us now turn to one of the most important, well-studied, and, for lack of a better word, modified structures of formal language theory – the finite automaton, which in its general principles were defined and explored by Alan Turing in [Tur37].

In short, a finite automaton is an idealised machine that can be in any of a finite set of *states*. It reads an input string of symbols, and, upon reading a symbol, moves from one state to another. Using this simple, mechanistic framework, we can achieve great complexity. As great, it turns out, as when using regular expressions (Kleene's Theorem [Kle51]). Let us formalise this:

Finite automaton A *finite (string) automaton* is a structure $A = (\Sigma, Q, q_0, F, \delta)$, where

- Σ is the *input alphabet*
- Q is the set of *states*
- $q_0 \in Q$ is the *initial state*
- $F \subset Q$ is the set of *final states*, and
- $\delta : (\Sigma \times Q) \to 2^Q$ is the *transition function*

A *configuration* of an automaton A is an element of $(Q \times \Sigma^*)$, that is, a state paired with a string or, equivalently, a string over $Q \cup \Sigma$ where only the first symbol is taken from Q. The *initial configuration* of A on a string w is the configuration q_0w, a *final configuration* is q_f, where $q_f \in F$, and a *run* of A on w is a sequence of configurations $q_0w = q_0w_0, q_1w_1, \ldots, q_kw_k = q_k$, where for each i, $w_i = cw_{i+1}$ for some $c \in \Sigma$, and $q_{i+1} \in \delta(q_i, c)$. If a run ends with a final configuration, then the run is *successful*, and if there is a successful run of an automaton on a string, the automaton *accepts* that string. The *language* of an automaton is the set of strings it accepts.

An automaton that accepts the finite language $\{$"a","aa","aba"$\}$ is, for example, $A = (\{a,b\}, \{q_0, q_1, q_2, q_f\}, q_0, \{q_1, q_f\}, \delta)$ where

$$\delta = \{(q_0, a) \to \{q_1\}, (q_1, a) \to \{q_f\}, (q_1, b) \to \{q_2\}, (q_2, a) \to \{q_f\}, \}$$

depicted in Figure 3.1, and one for the language "all strings with an even number of a's in Figure 3.2.

As mentioned, there are many well-known and well-studied modifications of the basic form of a finite state automaton,[2] such as augmenting the processing with an unbounded stack (yielding push-down automata), or an unbounded read-write tape

[2] Finite automata are perhaps more correctly described as being derived by restricting the more general Turing machines, which was the initially defined machine given in [Tur37].

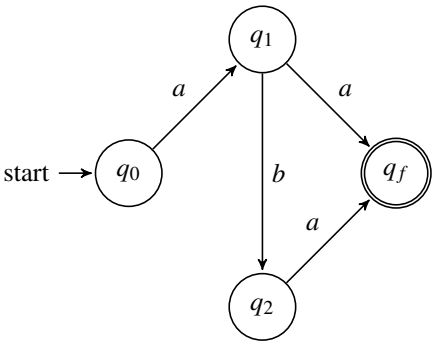

Figure 3.1: An automaton for the language {"a","aa","aba"}.

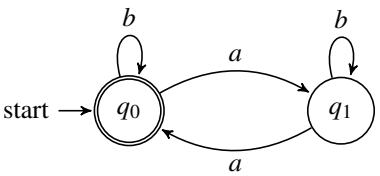

Figure 3.2: An automaton for the language of all strings over $\{a,b\}$ with an even number of a's

(yielding Turing machines), or restricting it, for example by forbidding loops (restricting the use to finite languages). Another common extension is to allow the automaton to write to an *output* tape, yielding string-to-string *transducers*

Another restriction which is more relevant to this thesis is to require the transition function be single-valued, i.e. instead of yielding a set of states, it yields a single state. The definition of runs is amended such that instead of requiring that $q_{i+1} \in \delta(q_i, c)$, we require that $q_{i+1} = \delta(q_i, c)$. Such an automaton is called *deterministic*, while ones defined as above are *nondeterministic*. Both recognise exactly the regular languages, but deterministic automata may require exponentially many more states to do so. Additionally, while nondeterministic automata may have several runs on the same string, deterministic automata have (at most) one.

3.2 Grammars

Finite automata is a formalism that works by recognising a given string, and regular expressions are well-specified descriptions of a languages. Grammars, in contrast, are *generative* descriptions of a languages. In general, grammars work by some kind of *replacement* of *nonterminals*, successively generating the next configuration, and ending up with a finished string of *terminals*. Though, again, many variants exist, and the practise of string replacement has a long history, the most studied and used type of grammar is the *context-free grammar*.

Context-free grammar A *context-free grammar* is a structure $G = (\Sigma, N, S, P)$ where

- Σ and N are finite alphabets of *terminals* and *nonterminals*, respectively
- $S \in N$ is the *initial nonterminal*, and
- P is a set of *productions*, on the form $A \to w$ where $A \in N$ and $w \in (\Sigma \cup N)^*$.

Intuitively, for a grammar, such as the context-free, we employ replacement by taking a string uAv and applying a production rule $A \to w$, replacing the left-hand side by the right-hand side, obtaining the new string uwv. The language of a grammar is the set of terminal strings that can be obtained by starting with the string S containing only the initial nonterminal, and then applying production rules until only terminal symbols remain.

Analogously to automata, we may call any string over $\Sigma \cup N$ a *configuration*. A pair w_i, w_{i+1} of configurations such that w_{i+1} is obtained from w_i by applying a production rule is called a *derivation step*. A sequence of configurations $w_0, w_1 \ldots, w_k$, starting with the string $w_0 = S$ containing only the initial nonterminal and ending with a string $w_k = w$ over Σ, where each pair w_i, w_{i+1} is a derivation step is a *derivation*.

While the context-free grammars are the most well-known and well-studied of the grammar formalisms, they do *not* correspond directly to regular expressions and finite automata in terms of computational capacity. That is, there are languages that can be defined by context-free grammars that no finite automaton recognises. Instead, CFG correspond to *push-down automata*, that is automata that have been augmented with a stack which can be written to and read from as part of the computation.

The grammar formalism that generates regular languages is named, unsurprisingly the *regular grammars*, and is defined in the same manner as the context-free, except that the strings on the right-hand side of productions are required to consist only of terminal symbols, with the exception of the *last* symbol. Without sacrificing any expressive power, we can even restrict the right-hand sides to be exactly aA where a is a terminal symbol and A is an optional nonterminal. If all rules in a regular grammars are on this form, we say that it is on *normal form*. Compare this to finite automata, where we start processing in one end and continue to the other, processing one symbol after the other.

The *parsing problem* for a certain class of grammars is the task of finding, given a grammar and a string, one or more derivation of the grammar that results in the string, if any exists. For regular grammars, it amounts, essentially, to restructuring the grammar into an automaton and running it on the string.

For a well-written introduction to many topics in theoretical computing science, and in particular grammars and automata on strings, see Sipser [Sip06].

3.3 Logic

Though the connection between language and logic is mostly thought of as concerning the semantic meaning of statements and analysing self-contradictions and implications in arguments, there is also a connection to formal languages. Let us first fix some notation for how to express logic.

We use $\neg$ for logical inversion, in addition to the standard logical binary connectives: $\wedge, \vee, \Leftrightarrow, \to$, as in Table 3.1

We can form logical *formulae* by combining facts (or *atoms*), with logical connectives, for example claiming that "It is raining" $\to$ "the pavement is wet", or "This is an ex-parrot" $\wedge$ "it has ceased to be". Each of these facts can be true or false, and

A	B	$A \wedge B$	$A \vee B$	$A \Leftrightarrow B$	$A \rightarrow B$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	F	T
F	F	F	F	T	T

Table 3.1: Standard logical connectives, T and F stands for "true" and "false", respectively

the truth of a formula is most often dependent on the truth of its component atoms. However, if we have some fact P, then the truth of the statements $P \vee \neg P$ and $P \wedge \neg P$ are both independent of P. We call the first formula (which is always true) a *tautology*, and the second (which is always false) a *contradiction*. Much has been written on this so-called *propositional logic*, both as a tool for clarifying arguments and as formal methods of proving properties of systems in various contexts.

We can be more precise about facts. Let us first fix a *domain* – a set of things that we can make statements about, such as "every human who ever lived", "the set of natural numbers", or "all strings over the alphabet Σ". We can then define subsets of the domain for which a certain fact is true, such as `even`, which is true of every even natural number, `dead`, which is true of all dead people, or `empty`, which is true of the empty string. A fact, then is for example `dead(socrates)`, which is true, or `dead(petter)`, which as of this writing is false.

We can generalise this precise notion of facts to not just be claims about the properties of single objects, but claims of *relations*. For example, we could have a binary `father` relation, which would be true for a pair of objects where the first is the father of the second, or a trinary `concatenation` relation, which is true if the first argument is the concatenation of the two following, as in `concatenation(aaab, a, aab)`. The facts about properties discussed previously are simply monadic relations.

A domain together with a set of relations (a *vocabulary*) and their definitions on the domain of objects is a logical *structure*, or *model*, and *model checking* is the practise of taking a model and a formula and checking whether or not the model *satisfies* the formula – that is, if the formula is true, given the facts provided by the model.

Up until now, we have discussed only propositional, or *zeroth-order* logic. Let us introduce *variables* and *quantification*, to yield *first-order* logic: We add to our logical symbols an infinite set $X = \{x, y, z \ldots\}$ of *variables*, disjoint from any domain, and the two symbols $\exists$ and $\forall$ that denote *existential* and *universal* quantification, respectively. Briefly, we write $\exists x\ \phi$ for some formula ϕ containing x to mean that "there exists some object x in the domain such that ϕ holds". Conversely, we write $\forall x\ \phi$ to mean "for *all* objects x in the domain, ϕ holds".

Quantification is thus a tool that allows us to express things like $\forall x\ \texttt{human}(x) \rightarrow (\texttt{alive}(x) \vee \texttt{dead}(x) \wedge \neg(\texttt{alive}(x) \wedge \texttt{dead}(x)))$, indicating that we have no vampires (who are neither dead nor alive) or zombies (who are both) in our domain, or at least that they do not count as human. Moreover, $\exists x\ \texttt{even}(x) \wedge \texttt{prime}(x)$ holds thanks to the number 2, assuming the domain is the set of natural numbers and `even` and `prime`

are given reasonable definitions.

Second-order logic uses the same vocabulary as first-order, but allows us to use variables not only for objects in the domain, but also for *relations*. Restricting ourselves to quantification over relations of arity one yields *monadic second-order* (MSO) logic, which is a type of logic with deep connections to the concept of regularity as defined using regular grammars or finite automata, see e.g. [Büc60].

Let our domain be the set of positions of a string, and define the relations $\texttt{lab}_a(x)$ for "the symbol at position x has the label a", and $\texttt{succ}(x,y)$ for "the position y comes directly after the position x". With these predicates, each string s has, essentially, one single reasonable logical structure $\mathscr{S}_s$ that encodes it. We can then use this encoding to define languages of strings, using logical formulae, saying that the language $\mathscr{L}(\phi)$ of a formula ϕ over the vocabulary of strings over Σ (i.e. using facts only on the form $\texttt{lab}_a(x)$ and $\texttt{succ}(x,y)$ for $a \in \Sigma$) is the set of strings s such that $\mathscr{S}_s$ satisfies ϕ.

For example, if we let $\phi = \forall x\,(\neg\exists z\,(\texttt{succ}(z,x)) \rightarrow \texttt{lab}_\texttt{a}(x))$, we capture all strings that start with an a in our language. We arrive at a new hierarchy of language classes, where first-order logic captures the *star-free* languages, and MSO logic captures the regular languages (Büchi's Theorem). The proof is somewhat technical, but for the direction of showing that all regular languages are MSO definable, it proceeds roughly as follows: We are going to assign each position of the string to a set, representing the various states that the automaton or grammar could have at that position. We do this by assigning the first position to the set representing the initial state, and then having formulae that represent transitions, checking that, for each pair of positions x,y, if $succ(x,y)$, then x is assigned to the proper set (say, $Q_q(x)$), that $\texttt{lab}_a(x)$, and then claiming that y is assigned to the proper set (say, $Q_{q'}(y)$). By finally checking whether or not the final position of the string belongs to any of the sets representing final states, we can ensure that the formula is only satisfied for structures $\mathscr{S}_s$ such that s is in the target language.

3.4 Composition and decomposition

We have thus far defined regularity in terms of automata, grammars and logic. We now introduce another perspective on regularity. First, let us define the universe of strings over an alphabet Σ, once again, this time algebraically. In short, the set of all strings over Σ is also named the *free monoid* over Σ, that is, the monoid with concatenation as the binary operation, and the empty string as the identity element.

We can moreover compose and decompose strings into *prefixes* and *suffixes*.

Given a string language L, we can define *prefix equivalence in relation to* L as the following: Two strings w and v are prefix equivalent in relation to L, denoted $\equiv_L$, if and only if for all strings $u \in \Sigma^*$, $uw \in L$ iff $uv \in L$. For each language, $\equiv_L$ is an equivalence relation on Σ^* – that is, it is a relation that is reflexive, symmetric and transitive. As such, it partitions Σ^* into equivalence classes, where all strings in an equivalence class are equivalent according to the relation. The number of equivalence classes for an equivalence relation is called its *index*, and we let the index of a language be the index of its prefix equivalence.

With these definitions in hand, we can define regular languages in a different way: The string languages with finite index are exactly the regular languages (Myhill-Nerode theorem) [Ner58].

3.5 Regularity, rationality and robustness

The regular languages have several other definitions and names in the literature, such as the *recognisable* languages, or the *rational languages*. What all these disparate definitions of regular languages have in common is that they are relatively simple, and for want of a better word, "natural", and though they come from radically different contexts (viz. algebra, formal grammars/automata, logic), they all describe the *same* class of languages. This is a somewhat unusual, though highly desirable property called *robustness*.

3.6 Learning of regular languages

Using finite automata or MSO logic, we can determine whether or not a string (or set of strings) belongs to a specific regular language. With regular grammars or expressions, we can, for a given language, generate strings taken from that language. However, sometimes we have no such formal description of a language, but we *do* have some set of strings we know are in the language, and some set of strings that are not. If we wish to *infer* or *learn* the language from these examples, we say we are trying to solve the problem of *grammatical inference* or *grammar induction*.

There are many different variants of this problem, such as learning only from positive examples or from a positive and negative set, these set(s) of examples being finite or infinite, having some coverage guarantees of the (finite) sets of examples, or having more or less control over which examples are given. The learning paradigm relevant to this thesis is one where not only is this control rather fine-grained, but in fact usually envisioned as a *teacher*, having already complete knowledge of the target language. More specifically, we are interested in the *minimally adequate teacher* (MAT) model of Angluin [Ang87], where the teacher can answer two types of queries:

- **Membership:** Is this string a member of the language?
- **Equivalence:** Does this grammar implement the target language correctly?

Membership queries are easily answered, but equivalence queries require not only an up-or-down boolean response, but a *counterexample*, that is, a string that is *misclassified* by the submitted grammar.[3]

Briefly, using membership and equivalence queries, the learner builds up a set of representative strings for each equivalence class of the target language, and a set of

[3] There are variants of MAT learning that avoids this type of equivalence queries using various techniques or guarantees on an initial set of positive examples.

distinguishing suffixes (or prefixes), such that for any two representatives w_1, w_2, there is some suffix s such that either $w_1 s$ is in the language while $w_2 s$ is not, or vice versa.

While many learning paradigms are limited to some subset of the regular languages, MAT learning can identify any regular language using a polynomial number of queries (in the number of equivalence classes of the target language).

3.7 Weights

We can augment our grammars and automata with *weights*,[4] yielding regular string series, more well known as recognisable series. There is a rich theory in abstract algebra with many results and viewpoints concerning recognisable series, though most of these are not relevant for this thesis. Refer to [DKV09] for a thorough introduction to the subject. Briefly, we augment our grammars and automata with a *weight function*, which gives each transition or production a weight taken from some semiring. The weight of a run or derivation is the product of the weights of all its constituent transitions/productions, and the weight of a string, the *sum* of all its runs/derivations. Deterministic weighted grammars and automata are those where for each string there is only a single run or derivation of non-zero weight.

[4] There are several candidates for putting weights on logical characterisations, e.g. [DG07], but thus far no obviously superior one.

CHAPTER 4
Tree languages

Though useful in many contexts, regular string languages are generally not sufficiently expressive to model natural human language. For this, we require at minimum context-free languages.[1] An additional gain from moving to the context-free languages is the ability to give some structure to the generation of strings – we can for example say that a *sentence* consists of a *noun phrase* and a *verb phrase*, and encode this in a context-free grammar using the rule $S \to NP\ VP$, and then have further rules that specify what exactly constitutes a noun or verb phrase. That is, we can give *syntactic rules*, and have them correlate meaningfully to our formal model of the natural language.

Of course, with the move from regular to context-free languages we lose a number of desirable formal properties, such as MSO definability and closure under complement and intersection, as well as the very simple linear-time parsing achievable using finite automata.

However, though context-free production rules and derivations are more closely related to how we tend to think of natural language syntax, working exclusively with the output strings is not necessarily sufficient. Ideally, we would like to reason not only about the strings, but about the syntactic structures themselves. To this end, let us define a *tree* over an alphabet Σ as being either

- a symbol a in Σ, or
- a formal expression $a[t_1, \ldots, t_k]$ where a is a symbol in Σ, and $t_1, \ldots, t_k$ are trees

A specific tree such as $a[b, c[d]]$ can also be shown as in Figure 4.1. We call the position of a in this case the *top* or *root*, while b and d are *leaves* that together make up the *bottom*, or *frontier*. We say that a is the *parent* of its *direct subtrees* b and $c[d]$. Further, $a[b, c[d]], b, c[d]$ and d are all the *subtrees* of $a[b, c[d]]$. Note that each tree is a subtree of itself, and each leaf is a subtree as well.

The *paths* in a tree is the set of sequences that start at the root and then move from parent to direct subtree down to some node. In the example that would be the set $\{a, ab, ac, acd\}$, optionally including the empty sequence.

[1] There are some known structures in natural language that even context-free languages are insufficient for. As the more expressive context-sensitive languages are prohibitively computationally expensive, there is an active search among several candidates for an appropriate formalism of *mildly context-sensitive languages*. See my licentiate thesis [Eri17] for my contributions to that field.

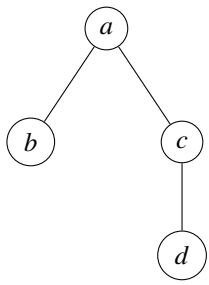

Figure 4.1: The tree $a[b, c[d]]$

4.1 Tree grammars

We can now let our CFG produce not strings, but trees, by changing each rule $A \to a_1 a_2 \dots a_k$ into $A \to A[a_1, a_2, \dots, a_k]$, with replacement working as in Figure 4.2. This gives us a *tree grammar* that instead of generating a language of strings generates a language of trees. In fact, these grammars are a slight restriction of *regular tree grammars*.[2] Unsurprisingly, many of the desirable properties that hold for regular string languages also hold for regular tree languages, but for technical reasons, this is easier to reason about using *ranked* trees, for which we require ranked alphabets:

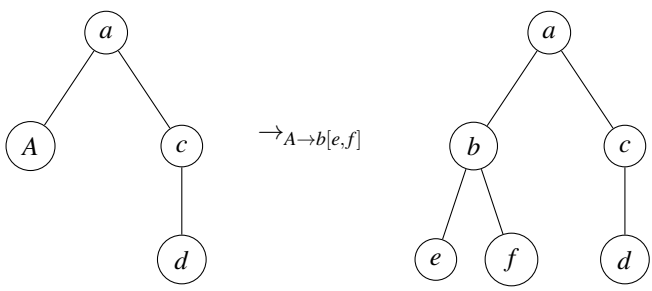

Figure 4.2: A tree replacement of A by $b[e, f]$.

Ranked alphabet A *ranked alphabet* $(\Sigma, rank)$ is an alphabet Σ together with a ranking function, $rank : \Sigma \to \mathbb{N}$, that gives a rank $rank(a)$ to each symbol in the alphabet. When clear from context, we identify $(\Sigma, rank)$ with Σ. We let Σ_k denote the largest subset of Σ such that $rank(a) = k$ for all $a \in \Sigma_k$, i.e. Σ_k is all the symbols $a \in \Sigma$ such that $rank(a) = k$.

We can now define the set T_Σ of ranked trees over the (ranked) alphabet Σ inductively as follows:

- $\Sigma_0 \in T_\Sigma$

[2] Specifically, over unranked trees.

- For $a \in \Sigma_k$ and $t_1 \dots t_k \in T_\Sigma$, $a[t_1, \dots, t_k] \in T_\Sigma$

We can immediately see that we can modify our previous CFG translations by letting all terminal symbols have rank 0 and creating several copies of each nonterminal A, one for each $k = |w|$ where $A \to w$ is a production. This also requires several copies of each production where A appears in the right-hand side. Though this may require an exponential number of new rules (in the maximum width of any right-hand side), the construction is relatively straightforward to prove correct and finite.

We now generalise our tree grammars slightly to obtain the regular tree grammars, as follows:

Regular tree grammar A *regular tree grammar* is a structure $G = (\Sigma, N, S, P)$ where

- Σ and N are finite ranked alphabets of *terminals* and *nonterminals*, respectively, where additionally $N = N_0$,
- $S \in N$ is the *initial nonterminal*, and
- P is a set of *productions*, on the form $A \to t$ where $A \in N$ and $t \in T_{(\Sigma \cup N)}$.

Now, the connection to context-free string grammars is obvious, but it is perhaps less obvious why these tree grammars are named *regular* rather than context-free. There are several ways of showing this, but let us stay in the realm of grammars for the moment.

Consider the way we would encode a string as a tree – likely we would let the first position be the root, and then construct a *monadic* tree, letting the string grow "downwards". The string *abcd* would become the tree $a[b[c[d]]]$, and some configuration of a regular string grammar *abcA* would become $a[b[c[A]]]$. Now, a configuration *abAc* of a *context-free* grammar would, by the same token, be encoded as the tree $a[b[A[c]]]$, but note that the nonterminal A would need to have rank 1 for this to happen – something which is disallowed by the definition of regular tree grammars, where $N = N_0$. This correlates to the restriction that regular grammars have only a single nonterminal at the very end of all right-hand sides.[3]

Another way to illustrate the connection is through the *path languages* of a tree grammar – the string language that is defined by the set of paths of any tree in the language. For monadic trees, this would coincide with the string translation used in the previous paragraph, but even allowing for wider trees, this is regular for any regular tree grammar. Moreover, as implied in the above sketch, each regular string language is the path language of some regular tree grammar.

4.2 Automata

As in the string case, the connection between automata and grammars is relatively straightforward. First we restrict the grammar to be on normal form, which in the tree case means that the right-hand sides should all be on the form $a[A_1, \dots, A_k]$ for $a \in \Sigma_k$ and $A_i \in N$ for all i.

[3] Context-free tree grammars, analogously, let nonterminals have any rank.

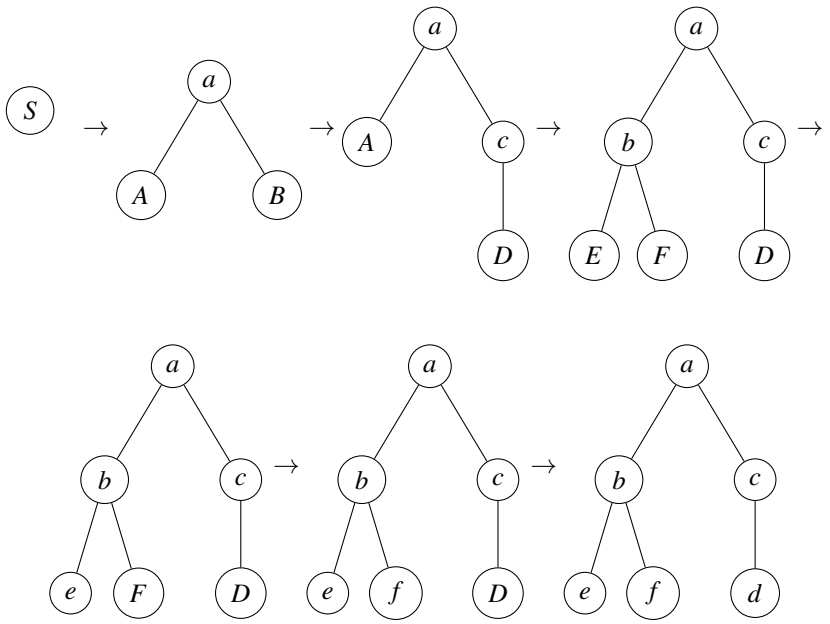

Figure 4.3: A derivation of a tree grammar.

Now, consider a derivation of a grammar on that form that results in a tree t, as in Figure 4.3. We wish to design a mechanism that, given the tree t, accepts or rejects it, based on similar principles as the grammar. Intuitively, we can either start at the bottom, and attempt to do a "backwards" generation, checking at each level if and how well a subtree matches a specific rule, or we start at the top and try to do a "generation" matching what we see in t. These are informal descriptions of *bottom-up* and *top-down finite tree automata*, respectively. Let us first formalise the former:

Bottom-up finite tree automaton A *bottom-up finite tree automaton* is a structure $A = (\Sigma, Q, F, \delta)$, where

- Σ is the ranked *input alphabet*
- Q is the set of *states*
- $F \subset Q$ is the set of *final states*, and
- $\delta : \bigcup_k (\Sigma_k \times Q^k) \to 2^Q$ is the *transition function*

In short, to compute the next state(s) working bottom-up through a symbol a of rank k, we take the k states $q_i, i \in \{1, 2, \ldots, k\}$ computed in its direct subtrees, and return $\delta(a, q_1, \ldots, q_k)$. Note that we have no initial state – instead the transition function for symbols of rank 0 (i.e. leaves) will take only the symbol as input and produce a set of states from only reading that. Compare this to the top-down case, defined next:

Top-down finite tree automaton A *top-down finite tree automaton* is a structure $A = (\Sigma, Q, q_0, \delta)$, where

- Σ is the ranked *input alphabet*
- Q is the set of *states*
- $q_0 \in Q$ is the *initial state*
- $\delta : \bigcup_k (\Sigma_k \times Q) \to 2^{Q^k}$ is the *transition function*

Here, we instead have an initial state q_0, but no *final states*, as the transition function, again for symbols of rank 0, will either be undefined (meaning no successful run could end thus), or go to the empty sequence of states λ.

Runs and derivations for tree automata naturally become more complex than for the string case, though configurations are, similar to the string case, simply trees over an extended alphabet with some restrictions. In particular, in derivations of regular string grammars there is at most one nonterminal symbol present in each configuration, making the next derivation step relatively obvious. For regular tree grammars, in contrast, there may in any given configuration be *several* nonterminals, any of which could be used for the next derivation step. Likewise, for string automata one simply proceeds along the string, while the computation proceeds in parallel in several different subtree in runs for both top-down and bottom-up tree automata. Let us for the moment ignore this difficulty, as for the unweighted case *the order is irrelevant*, and the run will be successful or not, the derivation result in the same tree *regardless* of what particular order we choose to compute or replace particular symbols in, as long as the computation or replacement is possible.

Bottom-up finite tree automata recognise exactly the class of regular tree languages in both its deterministic and nondeterministic modes, but for top-down finite tree automata, only the nondeterministic variant does so. This asymmetry comes from, essentially, the inability of top-down deterministic finite tree automata to define the language $\{f[a,b], f[b,a]\}$, as it can designate that it expects either an a in the left subtree and a b in the right, or vice versa, but not both at the same time without also expecting $f[a,a]$ or $f[b,b]$. For a slightly aged, but still excellent introduction to the basics of tree formalisms, including this and many other results, see [Eng75].

4.3 Logic

Logic over trees works in much the same way as logic over strings – we let the domain be the set of positions in the tree, and the relations be, essentially, the successor relation, but with several successors. More specifically, let the vocabulary for a tree over the ranked alphabet Σ with maximum rank k be, for each $a \in \Sigma$, the unary relation $\mathtt{lab}_a$, and for each $i \in \{1, 2, \ldots, k\}$, the binary relation $\mathtt{succ}_i$.

It should come as no surprise that the set of MSO definable tree languages are exactly the regular tree languages [TW68]. As for the string case, the proof is quite technical, but is built on similar principles that apply equally well.

4.4 Composition and decomposition

Translating prefixes, suffixes and algebraic language descriptions to the tree case is slightly more complex than the relatively straightforward extension of grammars, automata and logic. In particular, while both prefixes and suffixes in the string case, for trees we must introduce the notion of *tree contexts*, which are trees "missing" some particular subtree. More formally, we introduce the symbol $\Box$ of rank 0, disjoint from any particular alphabet Σ under discussion, and say that the set C_Σ of *contexts* over the alphabet Σ is the set of trees in $T_{\Sigma\cup\{\Box\}}$ such that $\Box$ occurs exactly once.

We can then define *tree concatenation* as the binary operation $concat : (C_\Sigma \times T_\Sigma) \to T_\Sigma$, written $s = concat(c,t) = c[t]$ where s is the tree obtained by replacing $\Box$ in c by t. As for string languages, this gives us the necessary tools to define *context equivalence relative to L* where L is a tree language: trees t and s are context equivalent in relation to L if $c[t] \in L$ iff $c[s] \in L$ for all $c \in C_\Sigma$, written $s \equiv_L t$. As in the string case, the regular tree languages are exactly those for which $\equiv_L$ has finite index.[4]

4.5 Weights

Augmenting the transitions and productions of tree automata and grammars with weights taken from some semiring, and computing weights in a similar manner to the string case yields *recognisable tree series*.[5] In the case where a tree has a single run or derivation, the computation is simple - multiply all the weights of all the transitions/productions used – but if we have some tree $f[a,b]$ with a derivation

$$S \to f[A,B] \to f[a,B] \to f[a,b]$$

then we could also derive the same tree as

$$S \to f[A,B] \to f[A,b] \to f[a,b]$$

.

However, considering exactly the same derivation steps have been made, with no interaction between the reordered parts, it would seem silly to count these derivations as distinct in order to compute the weight of the tree.

We can formalise the distinction between necessarily different derivations and those that differ only by irrelevant reorderings by placing our derivation steps into a *derivation tree*. This is a special kind of ranked tree where the alphabet is the set of productions of a grammar, and the rank of a production $A \to t$ is the number ℓ of nonterminals in t. We call this the *arity* of the production. A derivation tree for a grammar $G = (\Sigma, N, S, P)$, then, is a tree over the ranked alphabet $(P, arity)$, where *arity* is the arity function that returns the arity of a rule. However, note that not *every* tree over this alphabet is a proper derivation trees, as we have more restrictions on derivations than just there being a nonterminal in the proper place.

[4] See [Koz92] for a quite fascinating history of, and accessible restatement and proof of this result.
[5] See, once again, [DKV09], specifically [FV09], for a more thorough examination of the topic.

Instead, a derivation tree is one where the labels of nonterminals is respected, in the sense that (i) the root is marked with some production that has the initial nonterminal S as its left-hand side, and (ii) for each subtree $(A \to t)[s_1, \ldots, s_\ell]$ such that the ℓ nonterminals in t are A_1 to A_ℓ we require that for each s_i, its root is $(A_i \to t_i)$ for $A_i \to t_i \in P$. The derivation tree of the example derivation would then be

$$(S \to f[A,B])[(A \to a),(B \to b)]$$

We then define the weight of a tree t according to a weighted regular tree grammar G to be sum of the weights of all its distinct derivation trees, where the weight of a derivation tree is the product of the weights of all its constituent productions.

CHAPTER 5

(Order-Preserving) Graph Languages

Shifting domains once again, let us discuss graphs and graph languages, with the background we have established for strings and trees. Graphs consist of *nodes* and *edges* that connect nodes. As such, graphs generalise trees, which in turn generalise strings. However, while the generalisation of regularity from strings to trees is relatively painless, several issues make the step to graphs significantly more difficult.

Let us first define what type of graphs we are focusing on. First of all, our graphs are *marked*, in the sense that we have a number of designated nodes that are *external* to the graph. Second, our graphs are *directed*, meaning that each edge orders its attached nodes in sequence. Third, our edges are not the standard edges that connect just a pair of nodes, but the more general *hyperedges*, that connect any (fixed) number of nodes. Fourth, our edges have *labels*, taken from some ranked alphabet, such that the rank of the label of each edge matches the number of attached nodes.

Thus, from now on, when we refer to graphs we refer to *marked, ranked, directed, edge-labelled hypergraphs*, and when we say edges, we generally refer to hyperedges.

Formally, we require some additional groundwork before being able to properly define our graphs. Let LAB, $\mathscr{V}$, and $\mathscr{E}$ be disjoint countably infinite supplies of labels, nodes, and edges, respectively. Moreover, for a set S, let $S^{\circledast}$ be the set of *non-repeating* strings over S, i.e. strings over S where each element in S occurs at most once. Let S^{+} be the set of strings over S excluding the empty string ε, and $S^{\oplus}$ be the same for non-repeating strings.

Marked, directed, edge-labelled hypergraph A *graph* over the ranked alphabet $\Sigma \subset LAB$ is a structure $g = (V, E, lab, att, ext)$ where

- $V \subset \mathscr{V}$ and $E \subset \mathscr{E}$ are the disjoint sets of *nodes* and *edges*, respectively
- $lab : E \to \Sigma$ is the *labelling*,
- $att : E \to V^{\oplus}$ is the *attachment*, with $rank(lab(e)) = |att(e)| - 1$ for all $e \in E$
- $ext : V^{\oplus}$ is the sequence of *external nodes*

Further, for $att(e) = vw$ with $v \in N$ and $w \in V^{\circledast}$ we write $src(e) = v$ and $tar = w$ and say that v is the *source* and w the sequence of *targets* of the edge e. This implies a

directionality on edges from the source to the targets. Graphs, likewise have sources and targets, denoted as $v = \dot{g}$ and $w = g_{\ddot{.}}$ for $ext = vw$. The rank $rank(x)$ of an edge or a graph x is the length of its sequence of targets. The *in-degree* of a node v is the size of the set $\{e : e \in E, v \in tar(e)\}$, and the *out-degree* the size of the set $\{e : e \in E, v = src(e)\}$. Nodes with out-degree 0 are leaves, and with in-degree 0, roots. We subscript the components and derived functions with the name of the graph they are part of, in cases where it may otherwise be unclear ($E_g, src_g(e)$ etc.).

Note that we in the above define edges to have *exactly* one source and any (fixed) number of targets. This particular form of graphs is convenient for representing trees and tree-like structures, as for example investigated by Habel et. al. in [HKP87]. Moreover, natural language applications tend to use trees or tree-like abstractions for many tasks.

Even with this somewhat specific form of graphs, however, we can immediately identify a number of issues that prevent us from defining a naturally "regular" formalism, most obviously the lack of a clear order of processing, both in the sense of "this comes before that, and can be processed in parallel", and as in "this must be processed before that, in sequence". The formalisms presented in this thesis solves both of these problems, but at the cost of imposing even further restrictions on the universe of graphs.

5.1 Hyperedge replacement grammars

We base our formalism on the well-known *hyperedge replacement grammars*, which are context-free grammars on graphs, where we, as in context-free string grammars, continuously replace atomic nonterminal items with larger structures until we arrive at an object that contains no more nonterminal items. While nonterminal items are single nonterminal symbols in the string case, now they are hyperedges labelled with nonterminal symbols. To this end, we partition our supply of labels LAB into countably infinite subsets LAB_T and LAB_N of terminal and nonterminal labels, respectively.

Hyperedge replacement grammar A *hyperedge replacement grammar (HRG)* is a structure $G = (\Sigma, N, S, P)$ where

- $\Sigma \subset LAB_T$ and $N \subset LAB_N$ are finite ranked alphabets of *terminals* and *nonterminals*, respectively
- $S \in N$ is the *initial nonterminal*, and
- P is a set of *productions*, on the form $A \to g$ where $A \in N$ and $rank(A) = rank(g)$.

Hyperedge replacement, written $g = h[\![e : f]\!]$, is exactly what it sounds like – given a graph h we replace a hyperedge $e \in E_h$ with a graph f, obtaining the new graph g by identifying the source and targets of the replaced edge with the source and targets of the graph we replace it with. See Figure 5.1 for an example. As in the string and tree

cases, replacing one edge with a new graph fragment according to a production rule is a derivation step. The set of terminal graphs we can derive using a sequence of derivations steps starting with just the initial nonterminal is the language of the grammar. For a thorough treatment of hyperedge replacement and HRG, see [DHK97].

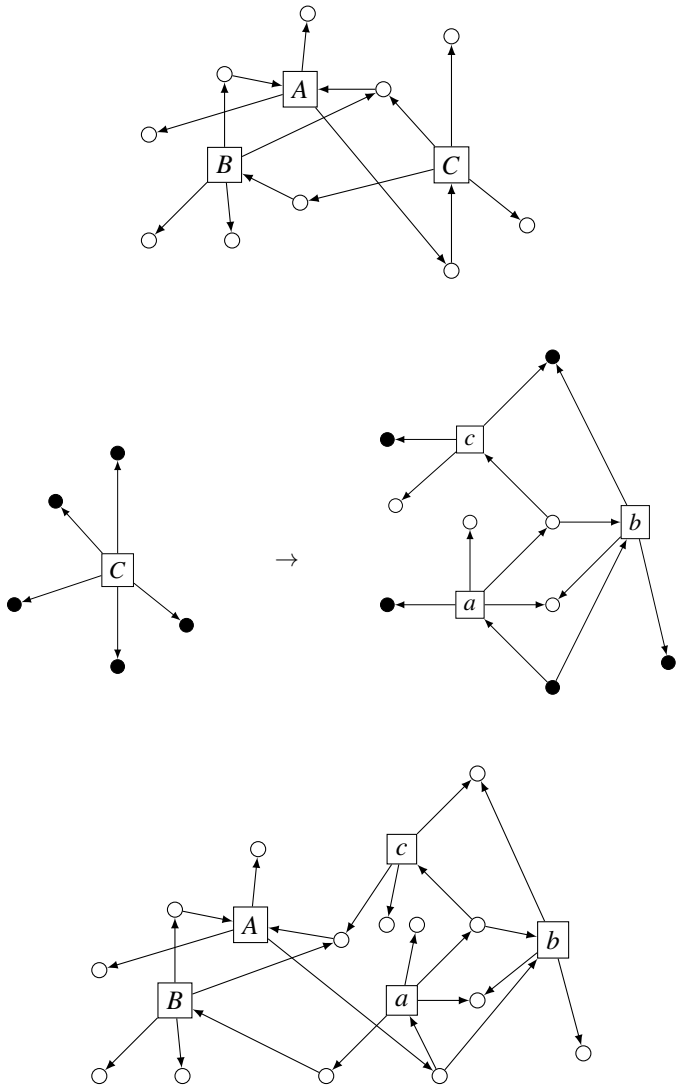

Figure 5.1: A hypergraph, replacement rule, and the result of applying the latter to the former

5.2 Issues of decomposition

A major difficulty in restricting HRG to something "regular" is, as mentioned, that there is no clear sense of "direction" in its derivations. Where strings go from one end to the other, and trees go from top to bottom (or bottom to top), we have no a priori obvious place in the right-hand sides of HRG to restrict our nonterminals to. Indeed, we have little sense of "position" and "direction" in graphs the first place, considering all the various connections that may occur. This is a major problem when it comes to efficient parsing of the graphs languages, as we would ideally like to be able to tell in "which end" to start processing, and how to choose a "next part" to continue with. Further, we would like these parts to be strictly nested, in the sense that we can immediately, from the structure of the graph, determine what larger subgraph of the graph the current subgraph is part of, in the same way that a specific subtree is not simultaneously a direct subtree of two different roots.

A natural first step in defining regular graph languages, then, is to simply decide that there is exactly one node that is the initial point from where everything starts generating, and that, moreover, the rest of the graph is at least *reachable* from that node. For us, the initial node is already clear – the source of the graph. Somewhat elided in the previous discussion on HRG, this remains the source of the graph even after replacing one of its nonterminals, and thus it is an ideal choice of a stable point.

In order to define reachability, let a *path* be a sequence $v_0, e_1, v_1, \ldots, e_k, v_k$ of nodes and edges such that for all $i \in \{1, 2, \ldots, k\}$, v_{i-1} is the source of e_i, and v_i is among the targets of e_i. We may optionally exclude the initial or the final node, or both. A path where $v_0 = v_k$ is a *cycle*, and path where v_0 is $\dot{g}$, a *source path*. Any node or edge on any path starting at a node v or edge e is *reachable from v (e)*. In a graph g with the source $\dot{g}$ we say that nodes or edges reachable from $\dot{g}$ are *reachable in g*. A graph g where $V_g \cup E_g$ are all reachable in g is *reachable*.

We now come to our first restriction on HRG, and on our universe of graphs – we require that all graphs are reachable.[1] Placing this restriction on all right-hand sides of our grammars ensures that all the graphs they generate will conform to this restriction as well (as the source of all nonterminals are reachable, so is the source of any graphs we replace them with, and these graphs in turn are all reachable from said source). Moreover, checking that an input graph has this property is easy.

The next restriction pertains to limiting the impact of a single nonterminal replacement. Consider a replacement like the one in Figure 5.2. From the final graph, there is no way to distinguish if the subgraphs having the external nodes as sources were part of the original graph, or if it was introduced in the replacement. For this reason, among others, we require that all targets of a right-hand side are leaves.

Together, these restriction provide an important property in that our grammars now are *path preserving* in the sense that if we have a replacement $g = h[\![e : f]\!]$ and two nodes $u, v \in V_h$, then u is reachable from v in g if and only if the same is true in h. Intuitively, this is because we no longer can introduce new paths between the targets of e, while all paths that pass $src_h(e)$ can still exit through the proper targets, as a result of us requiring f (and thus $f_{\bullet\bullet}$) to be reachable. Note that this property does *not*

[1] This also requires that the graphs are all connected.

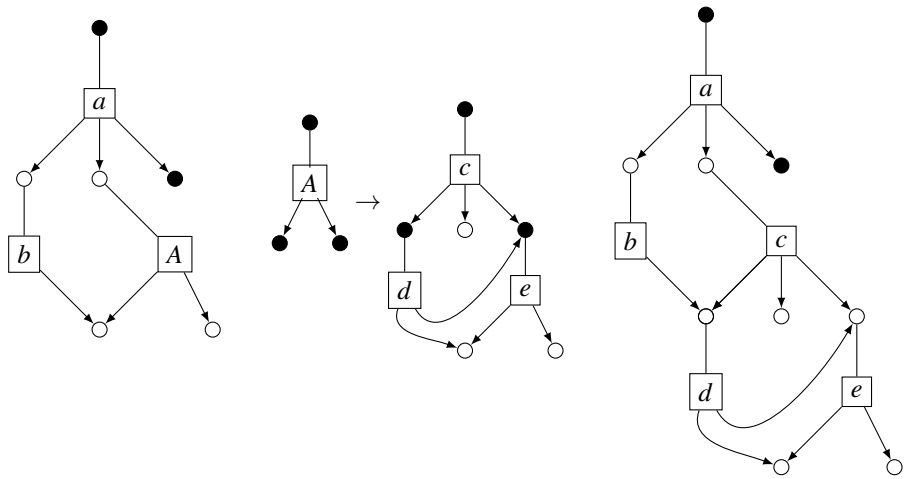

Figure 5.2: A hypergraph, replacement rule, and the result of applying the latter to the former

hold for all nodes in f, as there may be paths and cycles involving any combination of nodes of ext_f, including $\dot{f}$.

5.2.1 Reentrancies

We come now to a central concept in our formalisms – that of *reentrant nodes*. Briefly, the reentrant nodes of an edge or node x are the "first" nodes that can be reached from the root by passing x, but also using some path *avoiding* x. Let us make this more specific:

Reentrant nodes Let g be a graph. For $x \in V_g \cup E_g$, let $\hat{x}$ be x if $x \in V$, and $src_g(x)$ if $x \in E_g$. Moreover, let E_g^x be the set of all reachable edges e such that all source paths to e pass x, and let $TAR_g(E)$ for $E \subset E_g$ be the set of all targets of edges in E.

The set of *reentrant nodes of x in g* is

$$reent_g(x) = (TAR_g(E_g^x) \setminus \{\hat{x}\}) \cap (TAR_g(E_g \setminus E_g^x) \cup ext_g)$$

Looking at the final graph in Figure 5.2, the reentrant nodes of the root, and of the edge marked a, is simply the external node that is the third target of that same edge. For the edge marked e, only its leftmost target is reentrant, while for all other edges, their sets of reentrant nodes coincide with their targets.

We now introduce our third restriction on the grammars: That for every nonterminal edge e, $tar_g(e) = reent_g(e)$. The change from the previous is relatively small – this essentially means that all targets of nonterminals either have in-degree greater than 1, or are external nodes (or both). However, with all these restriction, our grammars are now *reentrancy preserving*,[2] in the following sense:

Reentrancy preservation Let $g = h[\![e : f]\!]$ be a replacement of a nonterminal e in h by f, resulting in g. The replacement is *reentrancy preserving* if, for all edges and nodes $x \in E_h \cup V_h$, $reent_g(x) = reent_h(x)$, and for all edges and nodes $x \in E_f \cup V_f \setminus ext_f$, $reent_g(x) = reent_f(x)$.

The precise claim is that if we have a grammar G with right-hand sides f conforming to the above restrictions, i.e.

- f is reachable
- All nodes in $f_{\bullet\bullet}$ are leaves
- For all nonterminal edges $e \in E_f$, $tar_f(e) = reent_f(e)$

Then all derivation steps of all derivations of G are reentrancy preserving. This is shown (with superficial differences) as Lemma 4.3 in Paper IV.

5.2.2 Subgraphs

Finally, with the above restrictions and concepts in place, we can define a hierarchy of subgraphs of each graph, delineated by each node and edge x and their reentrancies (see Lemma 3.4 of Paper IV). Moreover, with a reentrancy preserving grammar, we know that these subgraphs will be stable under derivation, and thus our parsing algorithm can assume that the reentrancies computed on the input will be useful throughout the parsing.

More formally, we let the *subgraph of g induced by x*, for $x \in V_g \cup E_g$ be the graph $g{\downarrow_x} = (E, V, lab, att, ext)$ where

- $E = E_g^x$
- $V = \{\hat{x}\} \cup TAR_g(E)$
- *lab* and *att* are the proper restrictions of lab_g and att_g to E, respectively
- *ext* is $\hat{x}$ followed by $reent_g(x)$ in some as of now unspecified order

Though a regular formalism is still some distance away, we now have an important set-piece: The ability to take an input graph and divide it into a tree-like hierarchy of subgraphs that can be processed one after the other, with clear lines between which graphs are direct subgraphs, and which are parallel lines of computation.

Let us look at a small example. In Figure 5.3 we have a small graph which has been generated by an OPDG. We have marked the "bottom-level" subgraphs using dashed, coloured lines. Note that these, as mentioned, can be found looking only at the structure of the input graph, with no reference to the grammar. The leftmost subgraph (indicated in green) has no reentrant nodes, and thus it must have come from a nonterminal of rank 0. In contrast the rightmost, indicated in purple, has two reentrant nodes, meaning that *it* must have come from a nonterminal of rank 2. Drawing in the "higher" subgraphs is left as an exercise for the reader.

[2] Note that this is not quite the same restrictions as used in Paper IV. Specifically, we lack restriction P2 of Definition 4.2, obviating the need for duplication rules, but this is not used in the relevant theorems and lemmas: Lemma 3.4, and Lemmas 4.3-4.5

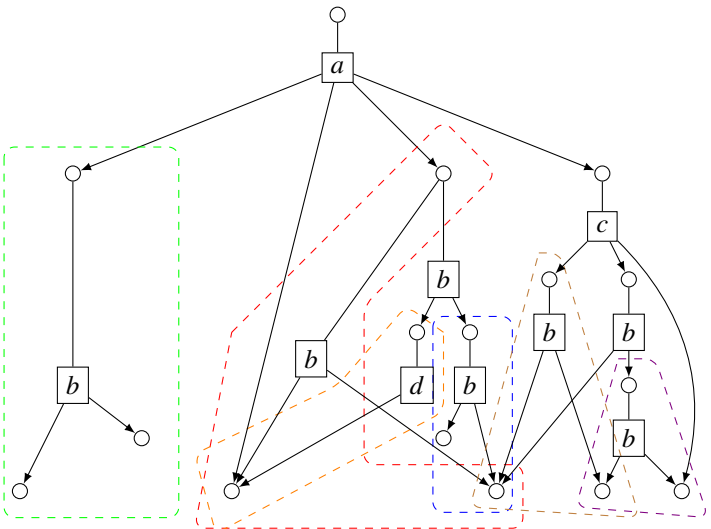

Figure 5.3: A graph with some of its subgraphs indicated.

5.3 Issues of order

An important consideration during the research that led to this thesis has been *efficient parsing*, rather than regularity itself. In particular, we have been searching for graph formalisms with efficient parsing in the *uniform* case. A thorough treatment of complexity theory is beyond the scope of this introduction,[3] but a major contributor to increased algorithmic complexity is the need to guess any particular node or edge, or even worse, a combination of nodes. Guessing the order of a set of nodes is another operation we wish to avoid, and in particular the need to remember an unbounded number of such guesses during the course of parsing.

Thus, while the hierarchy of subgraphs we have defined above is a *necessary* ingredient for efficient parsing, it is not yet *sufficient*.

In Paper IV, we investigate a set of requirements that lead to efficient parsing. One part is the reentrancy preservation that we have already discussed. The other is the property of *order preservation*. In the paper, this is a highly generalised set of requirements on the order itself (it needs to be efficiently computable, and order the targets of nonterminals according to the attachment), combined with a restriction on the rules that they preserve said order in a similar sense as the above on reentrancy preservation;

Order preservation Let $g = h[\![e : f]\!]$ be a replacement of a nonterminal e in h by f, resulting in g and let $\preceq_f, \preceq_h$ and $\preceq_g$ be ordering relations on the nodes of f, h and g, respectively. The replacement is *order preserving* if the restriction of $\preceq_g$ to V_f (V_h) is equal to $\preceq_f$ ($\preceq_h$).

[3] See e.g. [Pap03] for a useful textbook on the topic.

This is a slight restatement of Definition 4.7 of Paper IV. Let us now assume that our grammar is order preserving and reentrancy preserving. With structure and the order of nodes stable and discernible over derivations, we lack only a way to disambiguate or disregard the ordering of *edges*. In particular, while edges impose an order on their targets, no such ordering is imposed by a node on the edges that have it as their source. For this reason, we choose to disregard rather than disambiguate the order of edges, when it is ambiguous or undefined.

The restrictions that we now impose are thus intended to make sure that if we have some node with out-degree greater than 1, then we can parse it without referring to the order among the edges that are sourced there. We accomplish this by splitting our supply of rules into two sets: One where we limit the out-degrees of nodes to at most 1, and one a set of *duplication rules* with two edges, where nonterminal labels and attachments are required to be identical for both edges.[4] In Section 5.1 of Paper IV we show how duplication rules are parsed efficiently.

Finally, with all these restrictions, we have arrived at the formalism – Order-Preserving Hyperedge Replacement Grammars (OPHG) – that we present in Paper IV, and further investigate in Paper V. More properly, it is a *family* of formalisms, each with its own order. We show one example in Section 6 of Paper IV. It is still not known to be "regular", but does have efficient parsing in the uniform case.

5.4 A Regular Order-Preserving Graph Grammar

OPHG generalise an even more restricted type of graph grammar that was the initial object of study for this thesis – Order-preserving DAG grammars (OPDG). These can be seen as regular tree grammars augmented with (i) the ability to form connections through reentrant leaf nodes, and (ii) the possibility to create parallel structures through "cloning" a nonterminal.

In order to reason about the connections between OPDG and RTG, we need to be able to see trees as graphs. To this end, let us define a transformation from trees that yield graphs with the same structure in the following way: For a tree $t = a[t_1, \ldots, t_k]$ we create a graph g with nodes v_a and v_{t_i} for each $i \in \{1, 2 \ldots, k\}$, an edge e_a with $att_g(e_a) = v_a, v_{t_1}, \ldots, v_{t_k}$, and $lab(e_a) = a$. We then repeat the construction for each subtree in a similar way (substituting v_{t_i} for v_a where appropriate). See Figure 5.4 for an example. Note that when implementing a "regular tree grammar" as a HRG with the right-hand sides transformed in this way, the nonterminals will still appear only at the bottom of these graphs.

With this in place, we define our two extensions: First, we allow for *clone rules*, i.e. duplication rules where the two right-hand side nonterminals have the same symbol as the left-hand side. Second, we allow for leaves as targets, and moreover allow for leaves (but only leaves) with in-degree more than 1. In the style of OPHG, we require that the targets of nonterminals be reentrant, which in this context means that they must be leaves, and either external or the target of some other edges.

[4] We also require that for duplication rules $A \to f$ where $rank(A) = rank(f)$, that both edges in f have the label A.

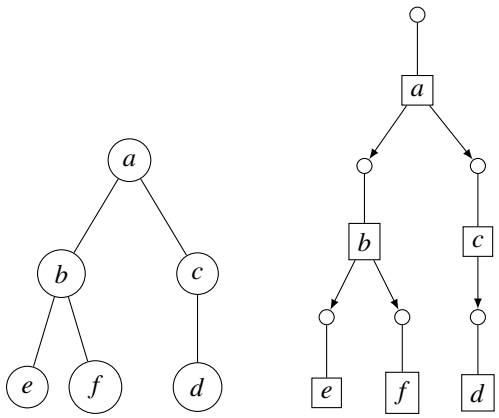

Figure 5.4: A tree and its translation into a graph.

Our grammars are still very close to regular tree grammars (with the exception of clone rules), and can be parsed strictly bottom-up in much the same fashion, as long as the order of leaves is accounted for. Let us look more closely at the way this is done in Papers I-III.

Closest common ancestor order If a node or edge x occurs on a source path ending at a node or edge y, we call x an *ancestor* of y, and y a *descendant* of x. If x is also an ancestor of z, it is a *common ancestor* of y and z. If no descendant of x is a common ancestor of y and z, then x is a *closest common ancestor*.

If x is an edge with targets $v_1, \ldots, v_k$ and it is a closest common ancestor of y and z, then there is a node v_i that is an ancestor of y, and moreover, that for all other nodes v_j that are ancestors of y, that $i < j$. Likewise we have a node v_k that is the first of the targets that are ancestors of z. If $i < k$ we say that *x orders y before z*.

We say that a graph g is *ccae-ordered* if, for all pairs of leaves y, z, all common ancestor edges of y and z either order y before z or z before y. We write $y \preceq_g z$.

Essentially, what we require from our right-hand sides f is that they are ccae-ordered, and that moreover, if a target y of f comes before another target z, that $y \preceq_f z$. This ensures that leaves are not reordered (or made to be unordered) by hyperedge replacement, and that all derivation steps are order-preserving for ccae-ordering. In Paper I (Theorem 8), we establish that disregarding ordering constraints leads to intractable parsing in the uniform setting.

5.4.1 Normal forms

In Paper I, we also show that we can put all OPDG into *normal form*, reminiscent of regular tree grammars – a normal form rule is either a clone rule, or contains a single terminal edge. As nonterminals are positioned only above leaves, this results in a graph consisting of a single terminal and an optional "layer" of nonterminals, with three "layers" of nodes – the source, the targets of the terminal, and the leaves.[5]

This, together with the close relation to regular tree grammars leads us to call OPDG a regular formalism. In particular, by "flipping" the arrow of normal form rules, we get something closely resembling tree automata, whereas this is not obviously the case for OPHG.

5.4.2 Composition and decomposition

Though the above discussion seems to indicate some sort of regular properties for OPDG, they can, in contrast to regular string and tree grammars, not generate the full universe of graphs, or even the universe of connected graphs. Indeed, even the universe of acyclic, single-rooted, reachable graphs, with only leaves having in-degree 1, is strictly larger than the union of all OPDG languages, due to cloning and ordering constraints.

The proper definition of the universe of graphs that OPDG *can* generate is somewhat technical, and is given in two different ways in Papers II and III. Briefly, we define a set of typed concatenation operators, each essentially matching a normal-form rule. Their input is a number of graphs, each with rank equal to the matching "nonterminal" in the operator, and the output is simply the graph obtained by replacing each such "nonterminal" with the corresponding input graph. The base case is all the graphs consisting of a single terminal symbol, possibly with some of its targets marked. The universe of graphs is the set of graphs that can be constructed using such concatenation operators.

With the universe thus defined, we again take inspiration from the tree case to define "prefixes" and "suffixes" – The universe of graph contexts over a specific alphabet is, as in the tree case, graphs "missing" a subgraph, which we can express as follows: We add to the set of "base" graphs the graphs $\square_k$ for each k, which consist of a single $\square$-labelled edge with its source and k targets, all external. The concatenation that constructs a graph context uses exactly one of these graphs in its construction.

Graph concatenation is only defined if the graph being inserted into the context has the same rank as the $\square$ edge, but is otherwise similar to the tree case. This includes the opportunity to define equivalence relative to some language for graphs, which unsurprisingly has finite index for OPDG languages. In Paper II we establish this fact, and use it to prove that OPDG are MAT learnable.

It is not known at this time if OPHG languages are MAT learnable.

5.4.3 Logic

Logic over graphs has two distinct implementations in the literature,[6] with quantification either over both nodes and edges or just over nodes, with edges implemented as relations in the structure. Using the former, we can define an MSO formula that identifies the graphs that conform to our OPDG restrictions. We show in Paper III that OPDG languages are MSO definable, but, crucially, not the reverse. That is, we have not shown that for every MSO formula that picks out a subset of our universe of graphs, we have an OPDG that defines the same language.

[5] Though this is an imperfect picture, as a node may be both a leaf and a target of the terminal.

[6] See e.g. pioneering work by Courcelle [Cou90]

This is a weakness in our claim that OPDG is a regular formalism, and one that we conjecture can be rectified in a future article. Future articles may also investigate the logical characterisation, if any, of OPHG.

5.5 Weights

As with the step from string to tree formalisms, when taking the step from tree grammars to OPDG and OPHG in terms of introducing weights, much is familiar, but some new complications are introduced, in particular in terms of what derivations are supposed to be seen as distinct for the purpose of the weight computation. Recall that we moved from linear derivations to derivation *trees*, which are trees over the ranked alphabet of *productions* with some additional restrictions. Two derivation trees are distinct if their root label differs or if any of their subtrees differ. For non-duplication rules, we can simply use this same framework, but as we explore in Papers III and V, this leads to unwanted outcomes for duplication rules.

More specifically, as both edges on the right-hand side of a duplication have the same source, and the same label, we do not care about their identity (i.e. which is e_1 and which e_2) for the purposes of keeping derivations distinct. In Papers III and V we make precise what this means for the equivalence of derivation trees for OPDG and OPHG, respectively.

Let us take a deeper look at what we actually mean by two derivations being distinct, which we can use to inform our understanding of weighted grammars more generally. For some entirely generalised grammar – not necessarily working with either strings, trees or graphs, let p_1 and p_2 be two productions. We, as usual, say that two derivation steps are *independent* if $x \to_{p_1} x' \to_{p_2} y$ and $x \to_{p_2} x'' \to_{p_1} y$. But by the discussion above on duplication rules, this is not sufficient for determining if two derivations are distinct, as this would, for example, not count duplicated edges as equivalent for the purposes of derivations.

Now, let us consider two full derivations $d_1 = x_0 \to_{p_1} x_1 \to_{p_2} x_2 \dots x_k$ and $d_2 = y_0 \to_{q_1} y_1 \to_{q_2} y_2 \dots \to_{q_k} y_k$ for the purpose of how to distinguish them. Generally, our notion of independent derivation steps lets us consider d_1 and d_2 equivalent if the independent derivation steps of d_2 can be rearranged to produce $d_2' = y_0 \to_{q_1'} y_1' \to_{q_2'} y_2' \dots \to_{q_k'} y_k$, where for each i, $y_i' = x_i$. If we instead change this requirement to be that for each i, y_i' is to *isomorphic* to x_i, our concept of distinct derivations more closely matches our intuition.

In particular, this new concept ensures that, for OPHG and OPDG derivations, which edge of a particular cloning we choose to apply further productions on is going to be irrelevant – the results will be isomorphic.

CHAPTER 6
Related work

OPDG and OPHG are far from the only attempts at defining or implementing grammars for working with semantic graphs in general, or even the specific one we have used to motivate our work – abstract meaning representations (AMR). Other parallel work has been looking at the specific theoretical properties we have attempted to achieve, such as polynomial parsing or a "natural" extension of the regular languages to graphs or DAGs. For semantic graphs, two major strains can be discerned in recent research; one focusing, like the present work, on hyperedge replacement and restrictions and extensions of same, and one focusing on various properties, restrictions and extensions of *DAG automata*.

6.1 Regular DAG languages

DAG automata define the regular DAG languages, and were originally introduced as straightforward generalisations of tree automata by Kamimura and Slutzki in [KS81]. They are usually defined as working on node-labelled directed acyclic graphs with node labels taken from a doubly ranked alphabet giving the in-degrees and out-degrees of the nodes. They have been further developed and investigated by Quernheim and Knight [QK12]. In [Chi+18], an investigation into the characteristics of regular DAG languages is made with an eye to semantic processing, with some encouraging, and some discouraging results. In particular, the parsing problem is shown to be NP-complete even in the non-uniform case, but a parsing algorithm is provided with relatively benign exponents.

Recent work by Vasiljeva, Gilroy and Lopez [VGL18] seems to indicate that DAG automata may be a suboptimal choice for weighted semantic models, if the intent is to have a probabilistic distribution. An interesting quirk of regular DAG languages is that they in general have regular path languages, but if the input graphs are required to be *single-rooted* this no longer holds. The specifics are explored in [BBD17]. See also [Dre+17] for a recent survey of results and open problems for DAG languages.

6.2 Unrestricted HRG

Major steps have been taken recently in making general HRG parsing more efficient in practise, see e.g. [Chi+13], and much seems to point to HRG being able to capture the structure of AMR in a reasonable way. However, numerous parameters in the input graphs – notably node degree – may contribute exponentially to the running time of the graph parsing. The same is also true of several grammar characteristics.

6.3 Regular Graph Grammars

In [Cou90; Cou91], Courcelle investigates several closely related classes of hypergraph languages – the languages of hyperedge replacement grammars (HRL),[1] the class of recognisable graph languages (REC), where the (context) equivalence relation has finite index, and the class of MSO definable graph languages (DEF). In particular, it is established that while DEF is properly included in REC, both are incomparable with HRL. Specifically, the languages that are recognisable but not in HRL are exactly the recognisable languages with unbounded treewidth.

The languages that are in the intersection of HRL and DEF are semantically defined as the *strongly context-free* graph languages, but a constructive definition is also given of a strongly context free class of graph languages: the *regular graph grammars*. This is reintroduced in [Gil+17] for use as an AMR formalism, and a parsing algorithm with slightly better, though still exponential complexity than for general HRG is given in [GLM17].

6.4 Predictive Top-Down HRG

Another restriction of HRG to obtain more favourable parsing complexity is *predictive top-down parsing* [DHM15], which is analogous to SLL(1) parsing for strings and uses *look-ahead* to limit the amount of backtracking that might be required during the parsing of a graph. The specific effects on the possible graphs and grammars is somewhat complex, and unfortunately [Jon16] seems to indicate that they may not be favourable for use with AMR.

6.5 S-Graph Grammars

S-graph grammars [Kol15] are able to produce the same graph languages as HRG, but does so in a quite different way, using an initial regular tree grammar to generate derivation trees for a constructive algebra. They have been applied to the AMR graph parsing problem and compared favourably to the unrestricted HRG parsing mentioned above. See [GKT15] for details.

[1] In [Cou91] HRL are referred to as the context-free graph languages, CF.

6.6 Contextual Hyperedge Replacement Grammars

While we argue that HRG are too powerful mainly for parsing complexity reasons, Drewes and Jonsson argue in [DJ17] that *extending* HRG to *contextual* HRG [DH15], while resulting in worse parsing complexity in general, captures more exactly and succinctly the actual structures of AMR over a set of concepts, thus resulting in more efficient parsing in practise.

CHAPTER 7
OPDG in Practise

The papers included in this thesis are almost exclusively built on theoretical results. To explore if these also carries practical usefulness, we conducted some very basic experiments, described below. Briefly, graphs were taken from two separate domains – abstract meaning representations and modular synthesis – and ingested into a hypergraph context with minimal transformations. In particular, the graphs were minimally and reversibly made to conform to the general structure of OPDG graphs, while order was ignored. Then, the order was checked for consistency according to ccae-order. Likewise, the sets of reentrant nodes for edges sharing a source were checked for equality. If a graph passes all these checks, then it can be generated by an OPDG, and if a large proportion of the graphs encountered in a bank are of that form, then we conjecture that a useful OPDG grammar could be inferred or constructed. The proportion of graphs where the order or reentrant sets was inconsistent is given below.

7.1 Abstract Meaning Representations

As briefly mentioned in the previous chapter, the development of OPHG and OPDG was motivated by *Abstract Meaning Representations (AMR)* [Ban+13]. These represent the semantics of actual sentences using graphs on the form shown in Figure 7.1. The construction of AMR graphs from sentences is currently done by humans, in order to build a graph bank of known correct examples which can be used as input for training a graph grammar. There is a lengthy specification [Ban+18] of how this is to be accomplished, which can be seen as an example of the "complex, rule-based" approaches mentioned above. However, note that this specification is intended to be used by *humans*, not computers, and thus it includes certain judgements that would otherwise be hard to encode.

The AMR process is thus, in some sense, a combination of the data-driven and the rule-based approaches mentioned at the end of Chapter 2 – use complex, hand-written rules in combination with human domain knowledge to build a large data bank, which then is used to train a relatively simple model. This is not unprecedented in NLP, as many syntactic treebanks have been built on the same principle, and for a similar purpose. The approach is less widespread in the context of semantics, however, with major projects starting mainly in the last decade.

We can immediately see from Figure 7.1 that both OPHG, and in particular our

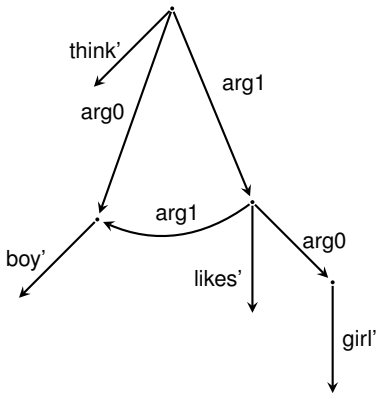

Figure 7.1: An AMR graph for "The boy thinks that the girl likes him"

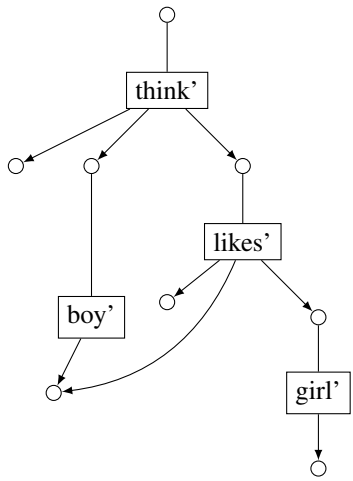

Figure 7.2: A "translation" of the AMR graph in Figure 7.1.

known learnable grammar OPDG may not be able to generate all AMR graphs (there are non-leaves with in-degree greater than 1, and no obvious control over the order). This is not ideal, as ostensibly, AMR is the motivation for their existence. However, let us take a closer look on the structure of an AMR.

We have a single root, and can assume that the graphs are acyclic. Each node has an identity, and at least one edge of rank 0 connected to it that shows what kind of thing or concept the node represents. There are also, depending on the type, a certain number of labelled edges pointing to arguments of whatever concept the node represents, and finally, potentially an unknown number of modifiers ("thinking deeply", is "think" modified with a "deeply" modifier). Our strategy, then, is to translate each such node into a hyperedge, labelled with the type, and with $k+2$ targets, where k is the number of arguments of the node. One of the extra nodes is used to collect all modifiers. For the case where we have a node of in-degree greater than 1, we choose one of its parents to be the primary one, and aim the other at the second extra target. In sum, the translation looks something like in Figure 7.2. Issues clearly remain, especially pertaining to order, but the experiments in Section 7.7 make this approach seem worthy of further study.

7.2 Modular synthesis

Briefly, a *modular synthesizer* is a synthesizer consisting of modules. That is, it is a mechanism for generating sounds and control signals from basic components such as sound sources, filters, sequencers, clock dividers, sample players, effects units, physical triggers, signal modifiers etc. Each module has a number of inputs and outputs, and these can be connected to each other in almost any configuration. The network

that defines a specific sound (a *patch*) can be seen as a hypergraph, with the modules being the hyperedges, and the connection points being the nodes.

7.3 Typed Nodes

In modular synthesis it is important to consider what *type* a particular connection point has – broadly, either audio or control – and further that no more than one thing connected to said point is an *output*, whereas all other connections are *inputs*. Likewise, we also have semantic types and categories in AMR that carry important information, such as the object argument of an action concept actually being capable of that action.

Placing these typed nodes in a single well-specified order, we can reduce the potential reordering in a graph derivation (or parsing) to be only between nodes of the same type. As shown in Paper I, this does not, unfortunately, reduce our parsing problem from being NP-complete, but the intuition is that such reorderings are rare, or in the worst case, with properly chosen types, relatively meaningless.

7.4 Primary and Secondary References

Though our formalism has been generalised to handle graphs where reentrancies occur not only in the leaves, but anywhere in the graph (subject to reentrancy and order preservation), the more limited OPDG are still significantly easier to work with in several respects. As such, it is useful to have techniques to arrive at a leaf-reentrant DAG from a general rooted graph.

A simple way to do this is, if given a general rooted graph, for each source v of a hyperedge e with in-degree greater than one, we can designate one single edge as the *primary* referrer to e (and thus v). This done, we create a new target v' of e, and replace v by v' in all other edges targeting it. We call these others *secondary* referrers. With minor modifications (making no edge a primary referrer to the root), this can create a leaf-reentrant DAG from any rooted graph.

For AMR this is especially simple as, in the syntax of AMR, the nodes are initially given as nodes of a tree, giving the primary references directly, while the secondary references are given by further connections among nodes in the AMR. In modular synthesis graphs it is not as clear-cut which connection should be given the distinction of being the primary one.

7.5 Marbles

The software used for the following practical tests is originally a Tree Automata Workbench, written by the author in the course of completing the Master's Thesis [Eri12]. It is a modular system written in Scala with some useful abstractions and general conventions suitable for implementing and experimenting with automata and grammars on trees. Extending the functionality to the types of graphs discussed in this thesis proved relatively straightforward.

7.6 Modular Synthesis Graphs

Axoloti [Tae18] is an open source hard- and software project for modular synthesis, where modules are implemented with typed in- and outputs, mostly specified in XML files. Patches are also specified in XML files, and are thus relatively easy to parse into a manipulable form.

As an open and available dataset for modular synths patches, we have used the user-generated `contrib` [TC18] repository of the Axoloti project, containing about 560 complete patches, which, separating subpatches to separate graphs yields around 1400 graphs.

7.6.1 Translation

In our tests we have assumed that any module not connected to a global output can be disregarded, as these will not impact the final sound or behaviour of the patch. We also create a new root hyperedge that is connected to all global outputs. This results in a rooted graph, which we than transform into a leaf-reentrant DAG by the above construction, choosing the first occurring edge as the primary referrer.

7.7 Semantic Graphs

As described in Chapter 7.1 Abstract Meaning Representations are graphs representing the semantics of a single sentence, and the main motivation for the research included in this thesis. The specification of AMRs in its actual syntax is particularly well suited to our formalism in that each graph is essentially described as a tree of nodes with either a name and type, or a reference to a node defined elsewhere. The specific dataset we have tested with is the freely available Little Prince corpus [al18].

7.7.1 Translation

The tree structure of AMR mentioned above gives us the relations of primary referrals in a natural way, while any additional references are secondary. No further translations have been made. In particular, no ordering attempts have been made based on the types mentioned above.

7.8 Results

Due to time and resource constraints, the experiments are rather limited both in scope and rigour. As a shorthand for whether a specific input graph is 'covered' by the OPDG formalism, we have chosen to make a naive ingestion into the system, and then testing if the resulting graph is (i) rooted and reachable, and (ii) ordered. If the graph is reachable from the designated root (generally true by construction), and if there are no two edges ordering two nodes (leaves) in opposing ways, then the graph is on a structure that can be generated by an OPDG. We also checked that all edges sharing a source had the same set of reentrant nodes.

For the AMR Little Prince corpus, this fails only in 30 out of 1562 sentences. However, there is a natural explanation for this low number: In order for a graph to qualify as unordered there need to be two nodes that have two separate edges as closest common ancestors, and this occurs only in 80 graphs in the whole corpus. In a way, this is an encouraging finding, as the reordering of nodes is by far the most obvious difficulty in applying our grammars to real-world data. Thus, even though the naive ordering has obvious difficulties with ordering nodes consistently, it is not necessarily a major problems, as it is not something that comes into play very often, at least in this limited AMR corpus. Moreover, recall the translation given above, where we create a new target v' of each edge e. If we disregard e for the purposes of ordering inconsistencies involving v', the amount of unordered AMR graphs in the Little Prince corpus drops to a scant 12.

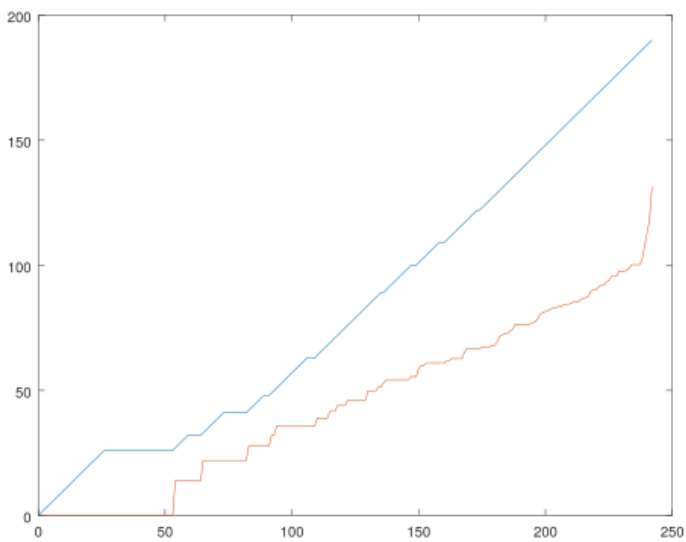

Figure 7.3: The cumulative sum of unordered graphs (blue), out of all potentially unordered modular synth graphs, ordered by number of potential unordered node pairs (red).

In the modular synthesis graph, ordering problems occur in 190 cases out of 1345 total tested graphs, that is, 14% of the graphs are unordered directly after the changes mentioned above. Here, only 242 graphs have of potential ordering problems, meaning that out of the potentially unordered graphs, around three fourths of them are actually unordered. This is likely due to there being no analogue to the tree structure in the specification of AMR, while the modules, in contrast, have a well-defined order on its connections in the Axoloti XML files already. As such, if one was to work more significantly with Axoloti graphs, it would be prudent to find some other way of or-

dering the nodes. A closer look at the statistics reveal further that, not unsurprisingly, graphs with few potential unorderings are less likely to be unordered than ones where many pairs of nodes have many shared closest common ancestor edges; see Figure 7.3, where the X axis represents the 242 graphs with potential reorderings, sorted by the amount of such reorderings. The blue line is the cumulative sum of unordered graphs. The red line is a scaled logarithm of the amount potentially unordered pairs of nodes in each graph. As can be plainly seen, almost every ordered graph is on the left side of the scale, i.e. has very few potential ordering problems.

There was no graph, in either corpus, where two edges sharing a source had different sets of reentrant nodes.

CHAPTER 8
Future work

The most immediate theoretical extension of the current work is to attempt to find corresponding proofs of "regular" properties for OPHG as already exists for OPDG, such as MSO definability, finite congruences for some reasonable definition, a Myhill-Nerode theorem, etc. For OPDG, similarly, a number of well-known results for regular tree languages should be relatively easy to prove for OPDG, such as various closedness and decidability results.

The most *interesting* direction to take for theoretical investigations may be to explore how well OPDG and OPHG match other semantic graph classes, as well as other graph representations. Another interesting topic is that of regularity for graphs, more generally – What *would* a regular formalism for graphs look like? Is it even possible to define, given the various difficulties we have encountered in doing so?

In terms of practical explorations, the first priority would be to properly implement both OPDG and OPHG into one or more systems suitable for testing. Second would be to do comparative testing against other already implemented systems for AMR parsing, and other graph parsing and generation problems.

Further practical explorations could focus on learning, either of weights in e.g. a expectation-minimisation model, or a practical implementation of a MAT oracle, either through human experts or some other model.

CHAPTER 9
Included Articles

9.1 Order-Preserving DAG Grammars

9.1.1 Paper I: Between a Rock and a Hard Place – Parsing for Hyperedge Replacement DAG Grammars

This paper is the first investigation into order-preservation, and thus uses various arguments, definitions and terminology that has been subsequently replaced by more elegant and descriptive terms. Nevertheless, the restrictions required for OPDG are introduced, together with a parsing algorithm. Additionally, a normal form is defined and proven to be relatively simple to construct for any given language. Furthermore, the restrictions of OPDG are motivated by giving several NP-completeness results for parsing variant grammars where one or more of the restrictions of OPDG are relaxed or removed. In particular, the ordering constraint is shown to be critical for polynomial uniform parsing.

9.1.2 Paper II: On the Regularity and Learnability of Ordered DAG Languages

Here we for the first time use the terminology of the rest of this thesis, and further develope the theory of OPDG. In particular, we define the universe of graphs in a way distinct from OPDG, and show that we have a Myhill-Nerode theorem for OPDG. This is used to instantiate an abstract MAT learner, which is proven correct using previous results. This algorithm is also a minimization algorithm for the deterministic unweighted case.

9.1.3 Paper III: Minimisation and Characterisation of Order-Preserving DAG Grammars

We deepen the connections between OPDG and regular tree grammars, showing their algebraic representations to be isomorphic. This is used to develop MAT learning, and thus minimisation, for OPDG in the weighted case. We also develop *concatenation schema*: a new way to conceive of the algebra generating the universe of graphs, as well as show OPDG to be MSO definable, giving additional credence to the idea of OPDG as a regular formalism. In addition, we replicate some results from Paper II using these new ideas.

9.2 Order-Preserving Hyperedge Replacement Grammars

9.2.1 Paper IV: Uniform Parsing for Hyperedge Replacement Grammars

The first paper on OPHG is also the first to properly identify the distinction between reentrancy preservation and order preservation, and to show the need for both for a uniformly polynomial parsing algorithm. Much of the paper is devoted to proving the nesting properties of reentrancies and subgraphs, and that OPHG are reentrancy preserving. The parsing algorithm is also defined and proven to run in polynomial time under certain conditions. We also give a concrete example of a suitable order, and additional restrictions under which OPHG preserve that order.

9.2.2 Paper V: Parsing Weighted Order-Preserving Hyperedge Replacement Grammars

The final paper builds mainly on results from Paper IV, introducing weights to the derivations of OPHG and making precise which derivations are to be seen as distinct for the purposes of computing weights for a graph. We also amend the parsing algorithm from Paper IV with weight computations and show it to be correct and remain efficient under the assumption that the weight operations are efficient.

9.3 Author's Contributions

Paper I: Conceptualization of formalism and parsing algorithm, translation of AMR

Paper II: Algebraic characterization, equality of graph classes, context definition

Paper III: Introducing weights, initial logical characterization

Paper IV: Generalisation parameters, separating reentrancy preservation from order preservation,

Paper V: Initial draft, derivation trees

CHAPTER 10

Articles not included in this Thesis

The papers described below were written during the course of the PhD program, though their content is significantly removed from the topic of order-preservation and graph grammars, and thus not included in this thesis.

10.1 Mildly Context-sentitive Languages

10.1.1 A Bottom-up Automaton for Tree Adjoining Languages

This technical report investigates the tree languages of the tree-adjoining grammars, in particular their path languages, and presents a bottom-up automaton for recognising them – essentially a regular bottom-up tree automaton augmented with an unbounded stack, and the opportunity to propagate the stack from any single subtree.

10.1.2 A Note on the Complexity of Deterministic Tree-walking Transducers

While tree adjoining languages are one of the weaker candidates for mildly context-sensitive languages, linear context-free rewriting systems (LCFRS) are one of the stronger. Here we investigate the precise parameterised complexities of LCFRS and an equally powerful formalism: deterministic tree-walking transducers. Unfortunately, the complexity results are for the most part negative.

Bibliography

[al18] Kevin Knight et al. *AMR Download page*. 2018. URL: https://amr.isi.edu/download.html (visited on 09/10/2018).

[Ang87] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Information and Computation* 75 (1987), pp. 87–106.

[Ban+13] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. "Abstract Meaning Representation for Sembanking". In: *Proc. 7th Linguistic Annotation Workshop, ACL 2013*. 2013.

[Ban+18] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, Hermjakobj U., K. Knight, P. Koehn, M. Palmer, and N. Schneider. *Abstract Meaning Representation (AMR) 1.2.5 Specification*. 2018. URL: https://github.com/amrisi/amr-guidelines/blob/master/amr.md (visited on 12/27/2018).

[BBD17] Martin Berglund, Henrik Björklund, and Frank Drewes. "Single-Rooted DAGs in Regular DAG Languages: Parikh Image and Path Languages". In: *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms*. 2017, pp. 94–101.

[Büc60] J Richard Büchi. "Weak second-order arithmetic and finite automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92.

[Chi+13] David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. "Parsing graphs with hyperedge replacement grammars". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1. 2013, pp. 924–932.

[Chi+18] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. "Weighted DAG automata for semantic graphs". In: *Computational Linguistics* 44.1 (2018), pp. 119–186.

[Cou90] Bruno Courcelle. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs". In: *Information and computation* 85.1 (1990), pp. 12–75.

[Cou91] Bruno Courcelle. "The monadic second-order logic of graphs V: On closing the gap between definability and recognizability". In: *Theoretical Computer Science* 80.2 (1991), pp. 153–202.

[DG07] Manfred Droste and Paul Gastin. “Weighted automata and weighted logics”. In: *Theoretical Computer Science* 380.1 (2007), p. 69.

[DH15] Frank Drewes and Berthold Hoffmann. “Contextual hyperedge replacement”. In: *Acta Informatica* 52.6 (2015), pp. 497–524.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. “Hyperedge Replacement Graph Grammars”. In: *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Ed. by G. Rozenberg. World Scientific, 1997. Chap. 2, pp. 95–162.

[DHM15] Frank Drewes, Berthold Hoffmann, and Mark Minas. “Predictive top-down parsing for hyperedge replacement grammars”. In: *International Conference on Graph Transformation*. Springer. 2015, pp. 19–34.

[DJ17] Frank Drewes and Anna Jonsson. “Contextual Hyperedge Replacement Grammars for Abstract Meaning Representations”. In: *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms*. 2017, pp. 102–111.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

[Dre+17] Frank Drewes et al. “On DAG languages and DAG transducers”. In: *Bulletin of EATCS* 1.121 (2017).

[Eng75] J Engelfriet. “Tree automata and tree grammars”. In: (1975).

[Eri12] Petter Ericson. *Prototyping the Tree Automata Workbench Marbles*. 2012.

[Eri17] Petter Ericson. “Complexity and expressiveness for formal structures in Natural Language Processing”. PhD thesis. Umeå Universitet, 2017.

[FV09] Zoltán Fülöp and Heiko Vogler. “Weighted tree automata and tree transducers”. In: *Handbook of Weighted Automata*. Springer, 2009, pp. 313–403.

[Gil+17] Sorcha Gilroy, Adam Lopez, Sebastian Maneth, and Pijus Simonaitis. “(Re)introducing Regular Graph Languages”. In: *Proceedings of the 15th Meeting on the Mathematics of Language*. 2017, pp. 100–113.

[GKT15] Jonas Groschwitz, Alexander Koller, and Christoph Teichmann. “Graph parsing with s-graph grammars”. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Vol. 1. 2015, pp. 1481–1490.

[GLM17] Sorcha Gilroy, Adam Lopez, and Sebastian Maneth. “Parsing Graphs with Regular Graph Grammars”. In: *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (* SEM 2017)*. 2017, pp. 199–208.

[HKP87] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. “Jungle evaluation”. In: *Workshop on the Specification of Abstract Data Types*. Springer. 1987, pp. 92–112.

[Jon16] Anna Jonsson. “Generation of abstract meaning representations by hyperedge replacement grammars–a case study”. MA thesis. Umeå University, 2016.

[Kle51] Stephen Cole Kleene. *Representation of events in nerve nets and finite automata*. Tech. rep. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[Kol15] Alexander Koller. “Semantic construction with graph grammars”. In: *Proceedings of the 11th International Conference on Computational Semantics*. 2015, pp. 228–238.

[Koz92] Dexter Kozen. “On the Myhill-Nerode theorem for trees”. In: *Bulletin of the EATCS* 47 (1992), pp. 170–173.

[KS81] Tsutomu Kamimura and Giora Slutzki. “Parallel and two-way automata on directed ordered acyclic graphs”. In: *Information and Control* 49.1 (1981), pp. 10–51.

[Ner58] Anil Nerode. “Linear automaton transformations”. In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544.

[NSS59] Allen Newell, John C Shaw, and Herbert A Simon. *Report on a general problem solving program*. Tech. rep. RAND PROJECT AIR FORCE SANTA MONICA CA, 1959.

[Pap03] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[QK12] Daniel Quernheim and Kevin Knight. “Towards probabilistic acceptors and transducers for feature structures”. In: *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics. 2012, pp. 76–85.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston, 2006.

[Tae18] Johannes Taelman. *Axoloti website*. 2018. URL: `http://www.axoloti.com/` (visited on 12/29/2018).

[TC18] Johannes Taelman and Axoloti Contributors. *Axoloti contrib repository*. 2018. URL: `https://github.com/axoloti/axoloti-contrib` (visited on 09/10/2018).

[Tur37] Alan M Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[TW68] James W. Thatcher and Jesse B. Wright. “Generalized finite automata theory with an application to a decision problem of second-order logic”. In: *Mathematical systems theory* 2.1 (1968), pp. 57–81.

[VGL18] Ieva Vasiljeva, Sorcha Gilroy, and Adam Lopez. “The problem with probabilistic DAG automata for semantic graphs”. In: *arXiv preprint arXiv:1810.12266* (2018).

[Wil68] John Wilkins. *An Essay Towards a Real Character and a Philosophical Language*. 1668.

I

Between a Rock and a Hard Place – Parsing for Hyperedge Replacement DAG Grammars

Henrik Björklund, Frank Drewes, Petter Ericson

Department of Computing Science, Umeå University, Sweden
{henrikb, drewes, pettter}@cs.umu.se

Abstract. We study the uniform membership problem for hyperedge-replacement grammars that generate directed acyclic graphs. The study of this type of language is motivated by applications in natural language processing. Our major result is a low-degree polynomial-time algorithm that solves the uniform membership problem for a restricted type of such grammars. We motivate the necessity of the restrictions by two different NP-completeness results.

1 Introduction

Hyperedge-replacement grammars (HRGs, see [7, 5]) are one of the most successful formal models for the generative specification of graph languages, thanks to the fact that their language-theoretic and algorithmic properties to a great extent resemble those of context-free grammars. Unfortunately, polynomial parsing is an exception from this general rule: graph languages generated by HRGs may be NP-complete. Thus, not only is the uniform membership problem intractable (unless P $\neq$ NP), but the non-uniform one is as well [1, 8].

Recently, Chiang et al. [4] advocated the use of hyperedge-replacement for describing meaning representations in natural language processing (NLP), and in particular the abstract meaning representations (AMRs) proposed by Banarescu et al. [2]. Chiang et al. described a general recognition algorithm building upon earlier work by Lautemann [9], together with a detailed complexity analysis. Unsurprisingly, the running time of the algorithm is exponential even in the non-uniform case, one of the exponents being the maximum degree of nodes in the input graph. Unfortunately, this is one of the parameters one would ideally not wish to limit, since AMRs may have unbounded node degree. However, AMRs and similar linguistic models to represent meaning are usually directed acyclic graphs (DAGs), a fact that is not exploited in [4]. Another recent approach to HRG parsing is [6], where predictive top-down parsing in the style of *SLL(1)* parsers is proposed. This is a uniform approach yielding parsers of quadratic running time in the size of the input graph, but the generation of the parser from the grammar is not guaranteed to run in polynomial time. (For a list of earlier attempts to HRG parsing, see [6].)

In this paper, we study the complexity of the membership problem for DAG-generating HRGs. Since NLP applications usually involve a machine learning component in which the rules of a grammar are inferred from a corpus, and hence the resulting HRG cannot be assumed to be given beforehand, we are mainly interested in efficient algorithms for the uniform membership problem. We propose restricted DAG-generating HRGs and show, in Section 4, that their uniform membership problem is solvable in polynomial time. More precisely, the upper bound on the running time of the algorithm is $\mathcal{O}(n^2+nm)$, where m and n are the sizes of the grammar and the input graph, resp. In linguistic applications, where grammars are usually much larger than the input structures to be parsed, this is essentially equivalent to $\mathcal{O}(nm)$. To our knowledge, this is the first time a uniform polynomial-time parsing algorithm for a non-trivial subclass of HRGs is proposed. Naturally, the restrictions are rather strong, but we shall briefly argue in Section 5 that they are reasonable in the context of AMRs. We furthermore motivate the restrictions with two NP-completeness results for DAG-generating HRGs, in Section 6. One of these proofs is a reduction of SAT to the uniform membership problem of DAG-generating HRGs whereas the second modifies the construction of [8] to show that there are NP-complete DAG languages of height 1 that can be generated by hyperedge replacement.

2 Preliminaries

The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ denotes $\{1, \dots, n\}$. Given a set S, let $S^{\circledast}$ be the set of non-repeating lists of elements of S. If $sw \in S^{\circledast}$ with $s \in S$, we shall also denote sw by (s, w). If $\preceq$ is a (partial) ordering of S, we say that $s_1 \cdots s_k \in S^{\circledast}$ *respects* $\preceq$ if $s_i \preceq s_j$ implies $i \leq j$.

2.1 Hypergraphs and DAGs

A *ranked alphabet* is a pair (Σ, rank) consisting of a finite set Σ of symbols and a *ranking function* $\mathrm{rank} : \Sigma \to \mathbb{N}$ which assigns a *rank* $\mathrm{rank}(a)$ to every symbol $a \in \Sigma$. We usually identify (Σ, rank) with Σ and keep the second component rank implicit.

Let Σ be a ranked alphabet. A (directed hyperedge-labeled) *hypergraph* over Σ is a tuple $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$ consisting of

- a finite set V of *nodes*,
- a *source* and *target mappings* $\mathrm{src}\colon E \to V$ and $\mathrm{tar}\colon E \to V^{\circledast}$ assigning to each hyperedge e its source $\mathrm{src}(e)$ and its sequence $\mathrm{tar}(e)$ of targets, and
- a *labeling* $\mathrm{lab}\colon E \to \Sigma$ such that $\mathrm{rank}(\mathrm{lab}(e)) = |\mathrm{tar}(e)|$ for every $e \in E$.

To simplify terminology, we shall in the following call hyperedges edges and hypergraphs graphs. Note that edges have only one source but several targets, similarly to the usual notion of term (hyper)graphs. The DAGs we shall consider below are, however, more general than term graphs in that nodes can have out-degree larger than one.

Continuing the formal definitions, a *path* in G is a (possibly empty) sequence $e_1, e_2, \ldots, e_k$ of edges such that for each $i \in [k-1]$ the source of e_{i+1} is a target of e_i. The *length* of a path is the number of edges it contains. A nonempty path is a *cycle* if the source of the first edge is a target of the last edge. If G does not contain any cycle then it is *acyclic* and is called a *DAG*. The *height* of a DAG G is the maximum length of any path in G. A node v is a *descendant* of a node u if $u = v$ or there is a nonempty path $e_1, \ldots, e_k$ in G such that $u = \mathrm{src}(e_1)$ and v occurs in $\mathrm{tar}(e_k)$. An edge e' is a *descendant edge* of an edge e if there is a path $e_1, \ldots, e_k$ in G such that $e_1 = e$ and $e_k = e'$.

The *in-degree* of a node $u \in V$ is the number of edges e such that u is a target of e. The *out-degree* of u is the number of edges e such that u is the source of e. A node with in-degree 0 is a *root* and a node with out-degree 0 is a *leaf*.

For a node u of a DAG $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$, the *sub-DAG rooted at* u is the DAG $G{\downarrow_u}$ induced by the descendants of u. Thus $G{\downarrow_u} = (U, E', \mathrm{src}', \mathrm{tar}', \mathrm{lab}')$ where U is the set of all descendants of u, $E' = \{e \in E \mid \mathrm{src}(e) \in U\}$, and src', tar', and lab' are the restrictions of src, tar and lab to E'. A leaf v of $G{\downarrow_u}$ is *reentrant* if there exists an edge $e \in E \setminus E'$ such that v occurs in $\mathrm{tar}(e)$.

2.2 DAG Grammars

A *marked* graph is a tuple $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab}, X)$ where $(V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$ is a graph and $X \in V^{\circledast}$ is nonempty. The sequence X is called the *marking* of G, and the nodes in X are referred to as *external nodes*. If $X = (v, w)$ for some $v \in V$ and $w \in V^{\circledast}$ then we denote them by $\mathrm{root}(G)$ and $\mathrm{ext}(G)$, resp. The former is motivated by the form or our rules, which is defined next.

Definition 1 (DAG grammar). *A* DAG grammar *is a system* $H = (\Sigma, N, S, P)$ *where* Σ *and* N *are disjoint ranked alphabets of* terminals *and* nonterminals*, respectively,* S *is the* starting nonterminal *with* $\mathrm{rank}(S) = 0$*, and* P *is a set of* productions*. Each production is of the form* $A \to F$ *where* $A \in N$ *and* F *is a marked DAG over* $\Sigma \cup N$ *with* $|\mathrm{ext}(F)| = \mathrm{rank}(A)$ *such that* $\mathrm{root}(F)$ *is the unique root of* F *and* $\mathrm{ext}(F)$ *contains only leaves of* F*.*

Naturally, a terminal (nonterminal) edge is an edge labeled by a terminal (nonterminal, resp.). We may sometimes just call them terminals and nonterminals if there is no danger of confusion. By convention, we use capital letters to denote nonterminals, and lowercase letters for terminal symbols.

A derivation step of H is described as follows. Let G be a graph with an edge e such that $\mathrm{lab}(e) = A$ and let $A \to F$ in P be a rule. Applying the rule involves replacing e with an unmarked copy of F in such a way that $\mathrm{src}(e)$ is identified with $\mathrm{root}(F)$ and for each $i \in [|\mathrm{tar}(e)|]$, the ith node in $\mathrm{tar}(e)$ is identified with the ith node in $\mathrm{ext}(F)$. Notice that $|\mathrm{tar}(e)| = |\mathrm{ext}(F)|$ by definition. If the resulting graph is G', we write $G \Rightarrow_H G'$. We write $G \Rightarrow_H^* G'$ if G' can be derived from G in zero or more derivation steps. The *language* $\mathcal{L}(H)$ of H are all graphs G over the terminal alphabet T such that $S^\bullet \Rightarrow_H^* G$ where $S^\bullet$ is the graph consisting of a single node and a single edge labeled by S.

The graphs produced by DAG grammars are connected, single-rooted, and as the name implies, acyclic. This can be proved in a straightforward manner by induction on the length of the derivation.

2.3 Ordering the Leaves of a DAG

Let $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$ be a DAG and let u and u' be leaves of G. We say that an edge e with $\mathrm{tar}(e) = w$ is a *common ancestor edge* of u and u' if there are t and t' in w such that u is a descendant of t and u' is a descendant of t'. If, in addition, there is no edge with its source in w that is a common ancestor edge of u and u', we say that e is a *closest* common ancestor edge of u and u'. We stress that since a node is a descendant of itself, this definition implies that if u and u' belong to w, then e is a closest common ancestor edge of u and u'. We also note that in a DAG, a pair of nodes can have more than one closest common ancestor edge.

Definition 2. *Let $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$ be a DAG. Then $\preceq_G$ is the partial order on the leaves of G defined by $u \preceq_G u'$ if, for every closest common ancestor edge e of u and u', $\mathrm{tar}(e)$ can be written as wtw' such that t is an ancestor of u and all ancestors of u' in $\mathrm{tar}(e)$ are in w'.*

3 Restricted DAG Grammars

DAG grammars are a special case of hyperedge-replacement grammars. We now define further restrictions that will allow polynomial time uniform parsing.

Every rule $A \to F$ of a *restricted DAG grammar* is required to satisfy the following conditions (in addition to the conditions formulated in Definition 1):

1. If a node v of F has in-degree larger than one, then v is a leaf
2. If F consists of exactly two edges e_1 and e_2, both labeled by A, such that $\mathrm{src}(e_1) = \mathrm{src}(e_2)$ and $\mathrm{tar}(e_1) = \mathrm{tar}(e_2)$ we call $A \to F$ a *clone rule*. Clone rules are the only rules in which a node can have out-degree larger than 1 and the only rules in which a nonterminal can have the root as its source.
3. For every nonterminal e in F, all nodes in $\mathrm{tar}(e)$ are leaves.
4. If a leaf of F has in-degree exactly one, then it is an external node or its unique incoming edge is terminal.
5. The leaves of F are totally ordered by $\preceq_F$ and $\mathrm{ext}(F)$ respects $\preceq_F$.

As is the case for DAG grammars in general, every graph that can be derived by a restricted DAG grammar is connected, single-rooted, and acyclic. We now demonstrate some additional properties.

Lemma 1. *Let $H = (\Sigma, N, S, P)$ be a restricted DAG grammar, G a DAG such that $S^\bullet \Rightarrow_H^* G$, and U the set of nodes of in-degree larger than 1 in G. Then U contains only leaves of G and $\mathrm{tar}(e) \in U^\circledast$ for every nonterminal e of G.*

Proof. We prove the lemma by induction. The base case, where $G = S^\bullet$ is immediate. Assume that G fulfils the conditions of the lemma and consider G' such that $G \Rightarrow_H G'$. Let $A \to F$ be the rule used in the derivation step.

By assumption, the edge e, labeled by A, that is rewritten has only leaves as targets. As nonterminals in F only appear directly above leaves in F and all the nodes in the marking of F are leaves, nonterminals of G' only appear directly above leaves.

Since only leaves have in-degree larger than 1 in G, all targets of A are leaves, and only leaves have in-degree larger than 1 in F, only leaves have in-degree larger than 1 in G'.

Since the edge that is being rewritten is nonterminal, it is not connected to any leaf with in-degree exactly 1. In F, leaves with in-degree exactly 1 are only connected to terminals. Thus the same holds in G'. □

3.1 Normal form

To simplify the presentation of parsing algorithm, we introduce a normal form for restricted DAG grammars.

Definition 3. *A restricted DAG grammar $H = (\Sigma, N, S, P)$ is on* normal form *if every rule $A \to F$ in P has one of the following three forms.*

(a) The rule is a clone rule.
(b) F has a single edge e, which is terminal.
(c) F has height 2, the unique edge e with $\mathrm{src}(e) = \mathrm{root}(F)$ is terminal, and all other edges are nonterminal.

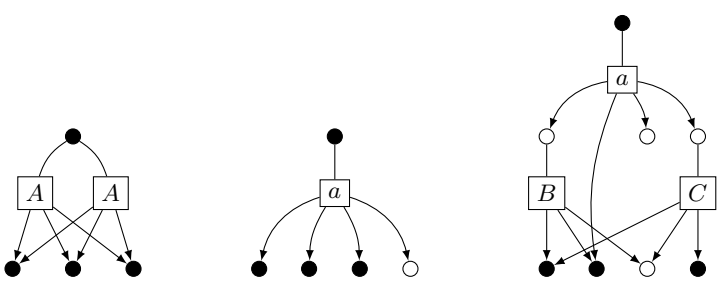

Fig. 1. Examples right-hand sides F of normal form rules of types (a), (b), and (c) for a nonterminal of rank 3. In illustrations such as these, boxes represent hyperedges e, where $\mathrm{src}(e)$ is indicated by a line and the nodes in $\mathrm{tar}(e)$ by arrows. Filled nodes represent the marking of F. Both $\mathrm{tar}(e)$ and $\mathrm{ext}(F)$ are drawn from left to right unless otherwise indicated by numbers.

See Figure 1 for examples of right-hand sides of the three types. In particular, right-hand sides F of the third type consist of nodes $v, v_1, \ldots, v_m, u_1, \ldots, u_n$, a terminal edge e and nonterminal edges $e_1, \ldots, e_k$ such that

– $v = \mathrm{root}(F) = \mathrm{src}(e)$ and $v_1 \cdots v_m$ is a subsequence of $\mathrm{tar}(e)$,
– $\mathrm{src}(e_i) \in \{v_1, \ldots, v_m\}$ for all $i \in [k]$,
– $\mathrm{ext}(F)$ and $\mathrm{tar}(e_i)$, for $i \in [k]$, are subsequences of $u_1 \cdots u_n$.

Lemma 2. *Every restricted DAG grammar H can be transformed in linear time into a restricted DAG grammar H' on normal form such that $\mathcal{L}(H) = \mathcal{L}(H')$.*

Proof. Let $H = (\Sigma, N, S, P)$ and let $r = A \to F$ be a rule in P. We present a recursive procedure for replacing r with a number of rules who together can derive F from A. If F has height 1, then due to restriction 2, r already has form (a) or (b). Thus, nothing needs to be done. Otherwise, we know that F has height at least 2 and, again by restriction 2, a unique edge e such that $\mathrm{src}(e) = \mathrm{root}(F)$. By the height of F, and since only leaves are targets of nonterminals, e is terminal.

Now, assume that F does not have the form (c). Then there exists a node v' in $\mathrm{tar}(e)$ which is not a leaf, such that the unique outgoing edge of v' is terminal. Let $F' = F{\downarrow_{v'}}$ and let s be the sequence of leaves in F, ordered according to $\preceq_F$. Notice that since no node in F has out-degree larger than 1, the leaves are totally ordered by $\preceq_F$, and $\mathrm{ext}(F)$ is a subsequence of s. Now, let s' be the subsequence of s consisting of the leaves in F' that are either in $\mathrm{ext}(F)$ or in $\mathrm{tar}(e')$ for an edge e' in F that does not belong to F'. We create a fresh nonterminal A' with $\mathrm{rank}(A') = |s'|$ and a rule $r' = A' \to (F', v's')$, i.e., the marking of the right-hand side is (v', s'). In F, we replace F' by A'. (More precisely, we remove all edges in F' from F, and likewise all nodes F' except for those in $v's'$, and we add a fresh edge f with $\mathrm{src}(f) = v'$, $\mathrm{tar}(f) = s'$, and $\mathrm{lab}(f) = A'$.)

Clearly, the language generated by the grammar is not affected by this decomposition of r into two rules. Moreover, each of the two new right-hand sides satisfies the conditions 1–5 and has fewer terminal hyperedges than F. Hence, by repeating the process we finally obtain an equivalent restricted DAG grammar in normal form. □

Lemma 3. *Let H be a restricted DAG grammar and $G = (V, E, \mathrm{src}, \mathrm{tar}, \mathrm{lab})$ a DAG generated by H. Then there is a total order $\trianglelefteq$ on the leaves of G such that $\preceq_G \subseteq \trianglelefteq$ and for every $v \in V$ and every pair u, u' of reentrant nodes of $G{\downarrow_v}$ we have $u \trianglelefteq u' \Leftrightarrow u \preceq_{G{\downarrow_v}} u'$.*

Proof. Note that it suffices to consider nodes v that are not leaves since the statement is trivially true if v is a leaf. Without loss of generality, we may furthermore assume that H is in normal form. We show by induction on the length of derivations that the statement holds for all DAGs G that can be derived from $S^\bullet$, not just the terminal ones. Moreover, we shall additionally prove that $\trianglelefteq$ can be chosen in such a way that $u_1 \trianglelefteq \cdots \trianglelefteq u_k$ for all nonterminals e in G with $\mathrm{tar}(e) = u_1 \cdots u_k$.

The DAG $S^\bullet$ has the claimed property as it does not possess any leaves. Now, consider a derivation $S^\bullet \Rightarrow^n G_0 \Rightarrow G$ and assume that the claim holds for G_0 with the total order $\trianglelefteq_0$. Let G be obtained from G_0 by applying a rule $r = A \to F$ to an edge e in G_0. There are three different cases to consider.

If the rule r is a clone rule, setting $\trianglelefteq = \trianglelefteq_0$ is sufficient because $\preceq_{G\downarrow_v} = \preceq_{G_0\downarrow_v}$ for all nodes v. This follows directly from the fact that the two edges e_1, e_2 that e is replaced with satisfy $\mathrm{tar}(e_1) = \mathrm{tar}(e) = \mathrm{tar}(e_2)$.

If r is of the form (b), let $\trianglelefteq$ be any total extension of $\trianglelefteq_0$ to the set of leaves of G that is consistent with $\preceq_F$. For all v, the reentrant nodes of $G\downarrow_v$ coincide with those of $G_0\downarrow_v$, and by restriction 5, $\preceq_{G\downarrow_v}$ coincides with $\preceq_{G_0\downarrow_v}$ on these nodes.

Finally, suppose r is of the form (c) and let $\mathrm{ext}(F) = v_1 \cdots v_k$. Leaves of F that are not in $\{v_1, \ldots, v_k\}$ and have in-degree 1 are not reentrant in any $G\downarrow_v$ and can thus be handled as in the preceding case, i.e., $\trianglelefteq_0$ can be extended to cover these nodes in any way that is consistent with $\preceq_F$. Let U be the remaining set of leaves of F, which thus includes $\{v_1, \ldots, v_k\}$. Since the nodes of F have out-degree at most one, U is totally ordered by $\preceq_F$, and by restriction 5 we have $v_1 \preceq_F \cdots \preceq_F v_k$. Moreover, by the induction hypothesis we may assume that $v_1 \trianglelefteq_0 \cdots \trianglelefteq_0 v_k$. We can thus extend $\trianglelefteq_0$ to a total order $\trianglelefteq$ on the leaves of G in such a way that the order coincides with $\preceq_F$ on U. It remains to argue that this definition of $\trianglelefteq$ has the claimed property.

To this end, let v be a non-leaf of G and let u, u' be reentrant nodes of $G\downarrow_v$. If v is a node in G_0 then u, u' are leaves of G_0 and we have

$$u \preceq_{G\downarrow_v} u' \Rightarrow u \preceq_{G_0\downarrow_v} u' \Rightarrow u \trianglelefteq_0 u' \Rightarrow u \trianglelefteq u'. \tag{1}$$

The remaining case is the one in which v is the source of a nonterminal edge f of F and u, u' are targets of f. If not both of u, u' are in $\mathrm{ext}(F)$ then f is the only closest common ancestor edge of u and u', and thus the claim immediately follows. If both u and u' are in $\mathrm{ext}(F)$ and u occurs before u' in $\mathrm{tar}(f)$, then $u \preceq_F u'$ by restriction 5. Consequently, $u \preceq_{G\downarrow_v} u'$ and also $u \preceq_{G_0\downarrow_v} u'$ because the only closest common ancestor of u and u' in G_0 that is not a closest common ancestor of them in G is f. Moreover, both u and u' are targets of f in G_0, so that the induction hypothesis yields $u \trianglelefteq_0 u'$. Altogether, we obtain the same chain of implications as in (1) above. □

3.2 Derivation Transparency

If a DAG G has been derived by a restricted DAG grammar in normal form, it is uniquely determined which subgraphs of G have been produced by a nonterminal, and which leaves were connected to it at that point. In particular, given a non-leaf node v in G, consider the subgraph $G\downarrow_v$. Consider the earliest point in the derivation where there was a nonterminal e having v as its source. We say that e generated $G\downarrow_v$. From the structure of G and $G\downarrow_v$, we know that all reentrant nodes of $G\downarrow_v$ are leaves and, by restriction 4, that e must have had exactly these reentrant leaves of $G\downarrow_v$ as targets. By Lemma 3 and restriction 5, the order of these leaves in $\mathrm{tar}(e)$ coincides with the total order $\preceq_{G\downarrow_v}$.

In other words, during the generation of G by a restricted DAG grammar, $G\downarrow_v$ must be generated from a nonterminal e such that $\mathrm{src}(e) = v$ and $\mathrm{tar}(e)$ is uniquely determined by the condition that it consists of exactly the reentrant

nodes of $G{\downarrow_v}$ and respects $\preceq_{G{\downarrow_v}}$. Therefore, we will from now on view $G{\downarrow_v}$ as a *marked* DAG, where the marking is $(v, \mathrm{tar}(e))$.

4 A Polynomial Time Algorithm

We present the parsing algorithm in pseudocode, after which we explain various subfunctions used therein. Intuitively, we work bottom-up on the graph in a manner resembling bottom-up finite-state tree automata, apart from where a node has out-degree greater than one. We assume that a total order $\trianglelefteq$ on the leaves of the input DAG G, as ensured by Lemma 3, is computed in a preprocessing step before the algorithm is executed. At the same time, the sequence w_v of external nodes of each sub-DAG $G{\downarrow_v}$ is computed. (Recall from the paragraph above that these are the reentrant leaves of $G{\downarrow_v}$, ordered according to $\preceq_{G{\downarrow_v}}$.) For a DAG G of size n, this can be done in time $O(n^2)$ by a bottom-up process. To explain how, let us denote the set of all leaves of $G{\downarrow_v}$ by U_v for every node v of G. We proceed as follows. For a leaf v, let $\trianglelefteq_v = \{(v, v)\}$ and $w_v = v$. For every edge e with $\mathrm{tar}(e) = u_1 \dots u_k$ such that u_i has already been processed for all $i \in [k]$, first check if $\trianglelefteq_0 = \bigcup_{i\in[k]} \trianglelefteq_{u_i}$ is a partial order. If so, define $\trianglelefteq_e$ to be the unique extension of $\trianglelefteq_0$ given as follows. Consider two nodes $u, u' \in U_{\mathrm{src}(e)}$ that are not ordered by $\trianglelefteq_0$. If i, j are the smallest indices such that $u \in U_{u_i}$ and $u' \in U_{u_j}$, then $u \trianglelefteq_e u'$ if $i < j$. Note that $\trianglelefteq_e$ is uniquely determined and total. Moreover, let w_e be the unique sequence in $U^{\circledast}_{\mathrm{src}(e)}$ which respects $\trianglelefteq_0$ and contains exactly the nodes in $U_{\mathrm{src}(e)}$ which are targets of edges of which e is not an ancestor edge. Similarly, if v is a node and all edges $e_1, \dots, e_k$ having v as their source have already been processed, check if $\bigcup_{i\in[k]} \trianglelefteq_{e_i}$ is a partial order. If so, define $\trianglelefteq_e$ to be any total extension of this order. Moreover, check that $w_{e_1} = \dots = w_{e_k}$, and let w_v be exactly this sequence.

After this preprocessing, Algorithm 1 can be run. As the sequences w_u of external nodes for each sub-DAG $G{\downarrow_u}$ were computed in the preprocessing step, we consider this information to be readily available in the pseudocode. This, together with the assumption that the DAG grammar H is in normal form allows for much simplification of the algorithm.

Walking through the algorithm step by step, we first extract the root node (line 2) and determine which kind of (sub-)graph we are dealing with (line 4): one with multiple outgoing edges from the root must have been produced by a cloning rule to be valid, meaning we can parse each constituent subgraph (line 5) recursively (line 6) and take the intersection of the resulting nonterminal edges (line 7). Each nonterminal that could have produced all the parsed subgraphs and has a cloning rule is entered into *returns* (line 8). The procedure `subgraphs_below` is used to partition the sub-DAG $G{\downarrow_v}$ into one sub-DAG per edge having v as its source, by taking each such edge and all its descendant edges (and all their source and target nodes) as the subgraph. Note that the order among these subgraphs is undefined, though they are all guaranteed by the preprocessing to have the same sequence of external nodes w_v.

Algorithm 1 Parsing of restricted graph grammars

```
 1: function PARSES_TO(restricted DAG grammar H in normal form, DAG G)
 2:     v ← root(G)
 3:     returns ← ∅
 4:     if out_degree(v) > 1 then
 5:         for G_i ← subgraphs_below(v) do
 6:             N_i ← parses_to(G_i)
 7:         N ← ⋂_i N_i
 8:         returns ← {A ∈ N | has_clone_rule(A)}
 9:     else
10:         e ← edge_below(v)
11:         children ← ()
12:         for v' ← targets(e) do
13:             if leaf(v') then
14:                 append(children, external_node(v'))
15:             else
16:                 append(children, parses_to(G↓_{v'}))
17:         returns ← {A | (A → F) ∈ P and match(F, e, children)}
18:     return returns
```

If, on the other hand, we have a single outgoing edge from the root node (line 9), we iterate through the subgraphs below the (unique) edge below the root node (line 12). Nodes are marked either with a set of nonterminals (that the subgraph below the nodes can parse to) (line 16), or, if the node is a leaf, with a boolean indicating whether or not the node is reentrant in the currently processed subgraph G (line 14).

The `match` function used in line 17 deserves a closer description, as much of the complexity calculations depend on this function taking no more than time linear in the size of the right-hand side graph on average. It works as follows:

Let $\mathrm{src}(e) = v$ and $\mathrm{tar}(e) = v_1 \cdots v_k$. Each v_i has an entry in *children*. If v_i is a leaf it is a Boolean, otherwise a set of nonterminal labels. From G and *children*, we create a DAG G' as follows. Let T be the union of $\{v, v_1, \ldots, v_k\}$ and the set of leaves ℓ of G such that ℓ is reentrant to G (as indicated by *children*) or there is an $i \in [k]$ with ℓ being external in $G{\downarrow_{v_i}}$. Let $T = \{v, v_1, \ldots, v_k, t_1, \ldots, t_p\}$. Then G' has the set of nodes $U = \{u, u_1, \ldots, u_k, s_1, \ldots, s_p\}$. Let h be the bijective mapping with $h(v) = u$ and $h(v_i) = u_i$ for every $i \in [k]$ and $h(t_i) = (s_i)$ for every $i \in [p]$. We extend h to sequences in the obvious way. The root of G is u and there is a single edge d connected to it such that $\mathrm{lab}(d) = lab(e)$, $\mathrm{src}(d) = u$ and $\mathrm{tar}(d) = u_1 \cdots u_k$. For every $i \in [k]$ such that v_i is not a leaf, G' has an edge d_i with $\mathrm{src}(d_i) = u_i$ and $\mathrm{tar}(d_i) = h(w_i)$, where w_i is the subsequence of leaves of $G{\downarrow_{v_i}}$ that belong to T, ordered by $\trianglelefteq$. The edge is labeled by the *set* of nonterminals $children[i]$.

Once `match` has built G' it tests whether there is a way of selecting exactly one label for each nonterminal edge in G' such that the resulting graph is isomorphic to *rhs*. This can be done in linear time since the leaves of both G' and *rhs* are

totally ordered and, furthermore, the ordering on $v_1 \cdots v_k$ and $u_1 \cdots u_k$ makes the matching unambiguous.

Let us now discuss the running time of Algorithm 1.

Entering the `if` branch of `parses_to`, we simply recurse into each subgraph and continue parsing. The actual computation in the `if`-clause is minor: an intersection of the l sets of nonterminals found.

Each time we reach the `else` clause in `parses_to`, we consume one terminal edge of the input graph. We recurse once for each terminal edge below this (no backtracking), so the parsing itself enters the `else`-clause n times, where n is the number of terminal edges in the input graph. For each rule $r = A \to F$, we build and compare at most $|F|$ nodes or edges in the `match` function. Thus, it takes $\mathcal{O}(nm)$ operations to execute Algorithm 1 in order to parse a graph with n terminal hyperedges according to a restricted DAG grammar H in normal form of size m. If H is not in normal form, Lemma 2 can be used to normalize it in linear time. Since the process does not affect the size of H by more than a (small) linear factor, the time bound is not affected. Finally, a very generous estimation of the running time of the preprocessing stage yields a bound of $\mathcal{O}(n^2)$, because n edges (and at most as many nodes) have to be processed, each one taking no more than n steps. Altogether, we have shown the following theorem, the main result of this paper.

Theorem 1. *The uniform membership problem for restricted DAG grammars is solvable in time $\mathcal{O}(n^2 + mn)$, where n is the size of the input graph and m is the size of the grammar.*

Note that in linguistic applications grammars are usually by orders of magnitude larger than the structures to be parsed (sentences, trees or, in our case, DAGs). Therefore, the bound given in Theorem 1 is essentially $\mathcal{O}(mn)$ in the context of such applications.

5 Representing and Generating AMRs

Let us have a very short glimpse at Abstract Meaning Representations (AMRs) and compare them with the type of DAGs considered in this paper. An AMR is an ordinary directed edge-labeled acyclic graph expressing the meaning of a sentence. An example expressing *"Anna's cat is missing her"* is shown in Figure 2. The root corresponds to the concept "missing", which takes two arguments, the misser and the missed.

In this representation every node has a special "instance edge" that determines the concept represented by its source node (miss, cat, anna). The most important concepts are connected to (specific meanings of) verbs, which have a number of mandatory arguments arg0, arg1, ... whose number depends on the concept in question. While the representation shown is not directly compatible with the restrictions introduced in Section 3 a simple translation helps. Every concept with its k mandatory arguments is turned into a hyperedge of rank $k + 1$, the target nodes of which represent the instance (a leaf) and the roots

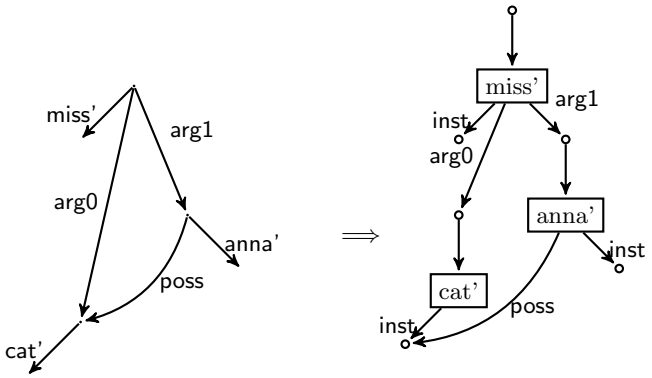

Fig. 2. Example translation of AMR.

of the arguments. The resulting hypergraph is shown in Figure 2 on the right. Note that all shared nodes on the left (corresponding to cross-references) are turned into reentrant leaves. This is important because in a DAG generated by a restricted DAG grammar only leaves can have an in-degree greater than 1.

It might seem that we only need graphs with nodes of out-degree at most 1, and thus no cloning rules for their generation. However, a concept such as miss can typically also have optional so-called modifiers, such as in *"Anna's cat is missing her heavily today"*, not illustrated in the figure. Such modifiers can typically occur in any number. We can add them to the structure by increasing the rank of miss by 1, thus providing the edge with another target v. The out-degree of this node v would be the number of modifiers of miss. Using the notation of Section 4, each sub-DAG $G{\downarrow_e}$ given by one of the outgoing edges e of v would represent one (perhaps complex) modifier. To generate these sub-DAGs $G{\downarrow_e}$ a restricted DAG grammar would use a nonterminal edge that has v as its source and which can be cloned. The latter makes it possible to generate any number of modifiers all of which can refer to the same shared concepts (represented by the leaves having the cloned nonterminals as their common targets).

On the generating side of AMRs, we immediately run into problems if the situation calls for multi-rooted graphs (e.g. two sentences connected via a conjunction or similar). Furthermore, the standard AMR solution for this situation (introducing a "dummy" root node, which connects to all the individual roots) is not necessarily applicable, as there might still be calls for connections among the different parts of the graph, which is a situation that cannot be covered by restricted DAG grammars. However, introducing a dummy *edge* above the different parts lets us decide on an order, and generate all the shared nodes beforehand, so to speak.

In Figure 3 we present a restricted DAG grammar that generates AMR-like graphs for all sentences consisting only of the concepts *boy*, *girl*, *want*, and *believe* in various combinations, an example that was introduced in [3]. Note that there

is only *one* boy and girl involved, which requires us to use a "dummy" root creating them (in order not to have several copies), along with the various subsentence start symbols.

The first row of rules constructs the basic structure of the graph – one edge each for *boy* and *girl*, and three basic statement edges. Any of these statement edges may be omitted, though we do not show these permutations in Figure 3. The second, third and fourth row are fairly self-explanatory. The rules for V_2

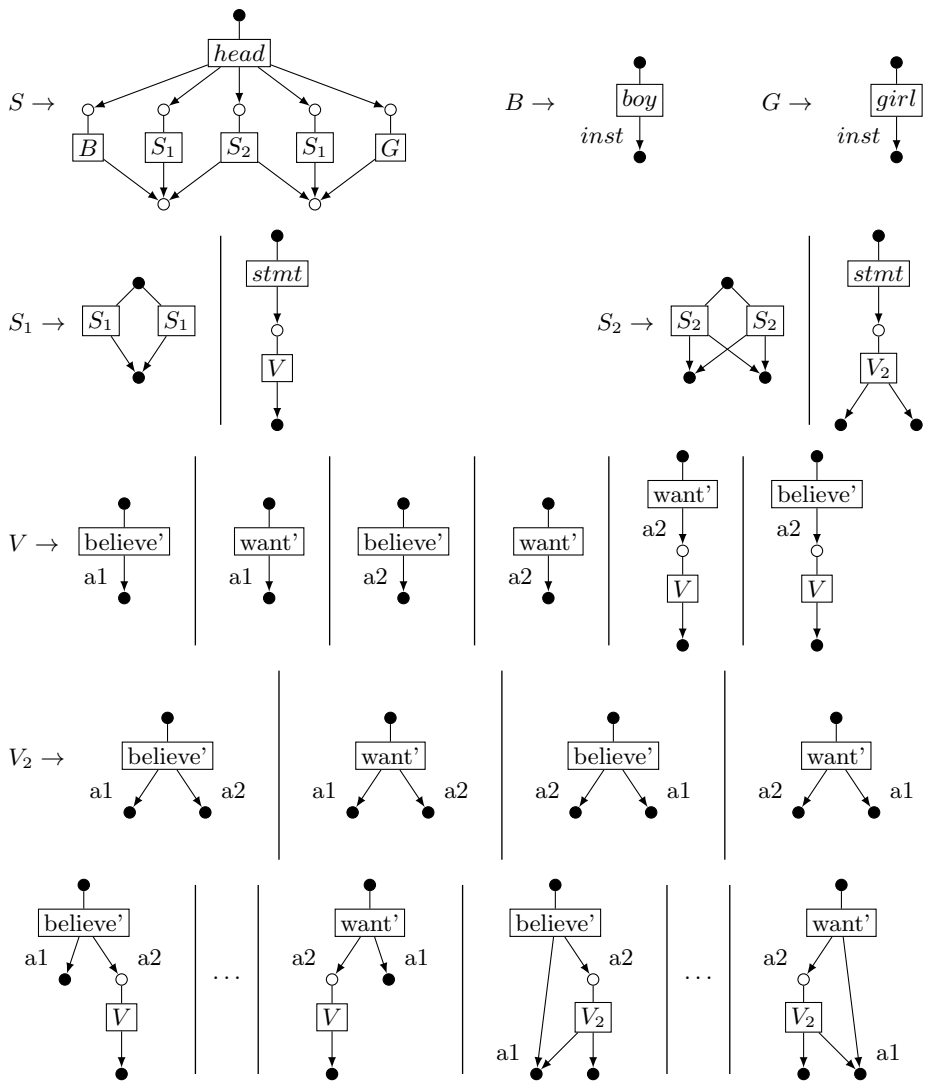

Fig. 3. Rules for a restricted DAG grammar generating AMR-like graphs for all sentences involving *boy*, *girl*, *want* and *believe*

involve quite a bit of (omitted) repetition. In particular, the first ellipsis cover two right-hand side graphs, the second another two.

Though the graph grammar is somewhat cumbersome, it serves as an example of a restricted DAG grammar generating a very general language of AMR-like graphs.

6 NP-hardness Results

In order to motivate the rather harsh restrictions we impose on our grammars, we present NP-hardness results for two different classes of grammars that are obtained by easing the restrictions in different ways.

Theorem 2. *The uniform membership problem for DAG grammars that conform to restrictions 1–4 is NP-complete.*

Proof. Clearly, the problem is in NP since the restrictions guarantee that derivations are of linear length in the size of the input graph. Thus, it remains to prove NP-hardness.

Let us consider an instance φ of the satisfiability problem SAT, i.e., a set $\{C_1, \ldots, C_m\}$ of clauses C_i, each being a set of literals x_j or $\neg x_j$, where $j \in [n]$ for some $m, n \in \mathbb{N}$. Recall that the question asked is whether there is an assignment of truth values to the variables x_j such that each clause contains a true literal. We have to show how to construct a DAG grammar H and an input graph G such that $G \in L(H)$ if and only if φ is satisfiable.

For simplicity, we shall first give a construction in which H violates conditions 4 and 5. The grammar uses nonterminals S, K, K_i, K_{ij} with $i \in [m], j \in [n]$. The terminal labels are c, all $j \in [m]$, and an "invisible" label. The labels K, K_i, K_{ij}, c are of rank $2n$, S is of rank 0 and the remaining ones are of rank 1. Figure 4 depicts the rules of the grammar. In this figure, and in the following, we draw ordinary edges (i.e., whose labels have rank 1) in the usual form as labeled arcs rather than boxes.

The grammar works in the following stages.

First row of rules: (1) Generate $2n$ leaves which, intuitively, represent $x_1, \neg x_1, \ldots, x_n, \neg x_n$ and are targets of a K-labeled nonterminal. (2) Clone K any number of times (where the intention is to clone it m times, once for each clause). (3) Let each K "guess" which clause C_i ($i \in \mathbb{N}$) it should check.

Second row of rules: (4) Let every K_i "guess" which literal makes C_i true. If the literal is negative, interchange the corresponding targets, otherwise keep their order.

Third row of rules: (5) For all pairs $(x_\ell, \neg x_\ell)$ that are not used to satisfy C_i, interchange the corresponding targets or keep their order. Finally, (6) replace the nonterminal edge by a terminal one.

Now, consider the input DAG G in Figure 5 (left). Suppose that G is indeed generated by H. Since the jth outgoing tentacles of all c-labeled edges point to the same node (representing either x_j or $\neg x_j$), a consistent assignment is

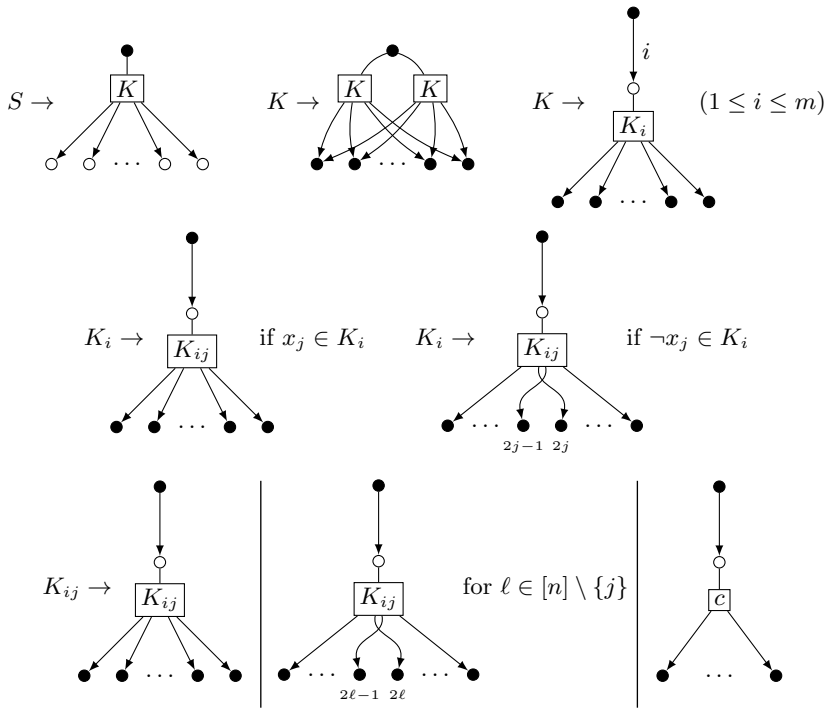

Fig. 4. Reduction of SAT to the uniform membership problem

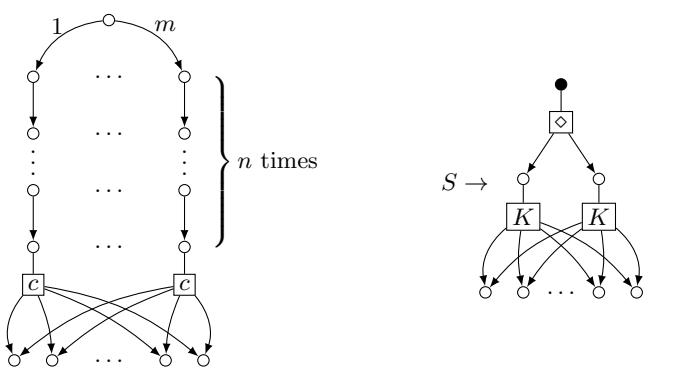

Fig. 5. Input graph in the proof of Theorem 2 (left) and modified starting rule (right)

obtained that satisfies φ. Conversely, a consistent assignment obviously gives rise to a corresponding derivation of G, thus showing that the reduction is correct.

Finally, let us note that changing the initial rule to the one shown in the left part of Figure 5 (using a new terminal $\diamond$ of rank 2) makes H satisfy condition 4 as well. This change being made, the input graph is changed by including two copies of the original input, both sharing their leaves, and adding a new root with an outgoing $\diamond$-hyperedge targeting the roots of the two copies. □

Let us now turn to our second NP-completeness result. It shows that if we, in addition, disregard restriction 2 in the definition of restricted DAG grammars, even the non-uniform membership problem becomes NP-complete. Moreover, this result holds even if all graphs generated by the grammar have height 1.

Theorem 3. *There is a DAG grammar H that conforms to restrictions 1, 3, and 4, such that all graphs in $\mathcal{L}(H)$ have height 1 and $\mathcal{L}(H)$ is NP-complete.*

Proof. The proof is by reduction from the (non-uniform) membership problem for *context-free grammars with disconnecting* (CFGD), using a result from [8]. A CFGD is an ordinary context-free grammar G in Chomsky normal form, with additional rules $A \to \diamond$, where $\diamond$ is a special symbol that cuts the string apart. Thus, an element in the generated language is a finite multiset of strings rather than a single string. More precisely, let $w = w_1 \diamond \cdots \diamond w_k \in (\Sigma \cup \{\diamond\})^*$, with $w_1, \ldots, w_k \in \Sigma^*$, be a string generated by G if we view G as an ordinary context-free grammar over $\Sigma \cup \{\diamond\}$. Then the multiset $\{w_1, \ldots, w_k\}$ is in $\mathcal{L}(G)$.

It is shown in [8] that CFGDs can generate NP-complete languages. Now, let us represent a multiset $\{w_1, \ldots, w_k\}$ of strings w_i as a graph consisting of k DAGs of height 1 sharing their roots, as follows. For a single string $w_i = a_1 \cdots a_m$, the graph $\mathrm{dag}(w_i)$ representing it consists of a root v, leaves $u_0, \ldots, u_m$, and a_i-hyperedges e_i with $\mathrm{src}(e_i) = r$ and $\mathrm{tar}(e_i) = u_{i-1}u_i$. Moreover, there are two terminal edges from v to u_0 and u_n, resp. (We draw the latter as unlabeled edges, using a special "invisible" label.) For a finite multiset $W = \{w_1, \ldots, w_k\}$ of strings w_i, $\mathrm{dag}(W)$ is obtained from the disjoint union of the individual DAGs $\mathrm{dag}(w_i)$ by identifying their roots. As an example, $\mathrm{dag}(\{ab, aba, c\})$ is shown in Figure 6.

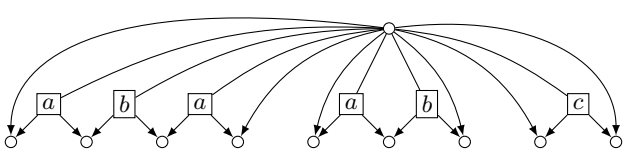

Fig. 6. The DAG $\mathrm{dag}(\{ab, aba, c\})$; note that the DAG does not define an order among the sub-DAGs $\mathrm{dag}(w_i)$ that constitute it

Now, every CFGD G can be turned into a DAG grammar H such that $\mathcal{L}(H) = \{\mathrm{dag}(W) \mid W \in \mathcal{L}(G)\}$ using the schemata in Figure 7. Hence, $\mathcal{L}(H)$ is NP-complete if $\mathcal{L}(G)$ is.

Though the simplicity of the translation should be sufficient to prove its correctness, a few remarks may be in order. On the one hand, the ordering of symbols within the representation of an individual string w_i is faithfully reflected in $\mathrm{dag}(w_i)$, due to the fact that $\mathrm{tar}(e)$ is an ordered sequence for each edge e, which unambiguously determines the start and end of the representation of w_i. On the other hand, there is no order among the represented strings in $\mathrm{dag}(W)$ as they are connected *only* via the root.

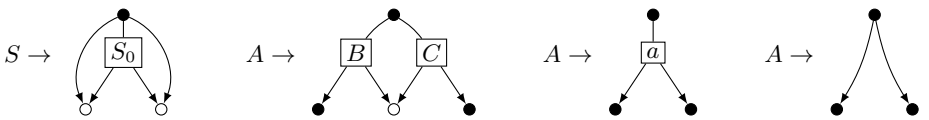

Fig. 7. Rules of a DAG grammar equivalent to a CFGD with initial nonterminal S_0, from left to right: initial rule, $A \to BC$, $A \to a$, $A \to \diamond$.

7 Conclusions

By enforcing rather severe restrictions, we have defined a class of hyperedge replacement graph grammars for which even the uniform parsing problem is solvable in low-degree polynomial time. We also argued that this class, despite its limitations, can still be practically relevant, e.g., in linguistic applications.

A number of interesting questions remain open. We motivate our restrictions by showing how two ways of easing them lead to NP-hardness, but this does not necessarily mean that all of our restrictions are necessary, neither does it mean that they are the only interesting ones. Is it the case that lifting any one of our five restrictions, while keeping the others, leads to NP-hardness? It seems that the algorithm we propose leads to a fixed-parameter tractable algorithm, with the size of right-hand sides in the grammar as the parameter, when we lift restriction 5 (enforcing that the marking respects $\preceq_F$). Is this actually the case and are there other interesting parameterizations that give tractability for some less restricted classes of grammars? Another open question is whether the algorithm for checking the structure of the input graph and computing the ordering on the leaves can be optimized to run in linear or $\mathcal{O}(n \log n)$ time.

From a practical point of view, one should study in detail how well suited restricted DAG grammars are for describing linguistic structures such as AMRs. Which phenomena can be modeled in an appropriate manner and which cannot? Are there important aspects in AMRs that can be modeled by general DAG-generating HRGs but not by restricted DAG grammars? If so, can the restrictions be weakened appropriately without sacrificing polynomial parsability?

References

1. I. J. Aalbersberg, A. Ehrenfeucht, and G. Rozenberg. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13:79–85, 1986.
2. L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. Abstract meaning representation for sembanking. In *Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop*, 2013.
3. F. Braune, D. Bauer, , and K. Knight. Mapping between english strings and reentrant semantic graphs. In *Proc. 9th Intl. Conf. on Language Resources and Evaluation (LREC'14)*, 2014.
4. D. Chiang, J. Andreas, D. Bauer, K. M. Hermann, B. Jones, and K. Knight. Parsing graphs with hyperedge replacement grammars. In *Proc. 51st Annual Meeting of the Association for Computational Linguistics (ACL 2013), Volume 1: Long Papers*, pages 924–932. The Association for Computer Linguistics, 2013.
5. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. World Scientific, Singapore, 1997.
6. F. Drewes, B. Hoffmann, and M. Minas. Predictive top-down parsing for hyperedge replacement grammars. In *Proc. 8th Intl. Conf. on Graph Transformation (ICGT'15)*, Lecture Notes in Computer Science. Springer, 2015.
7. A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.
8. K.-J. Lange and E. Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30, 1987.
9. C. Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.

II

On the Regularity and Learnability of Ordered DAG Languages

Henrik Björklund, Johanna Björklund, and Petter Ericson

Dept. Computing Science, Umeå University

Abstract. Order-Preserving DAG Grammars (OPDGs) is a subclass of Hyper-Edge Replacement Grammars that can be parsed in polynomial time. Their associated class of languages is known as Ordered DAG Languages, and the graphs they generate are characterised by being acyclic, rooted, and having a natural order on their nodes. OPDGs are useful in natural-language processing to model abstract meaning representations. We state and prove a Myhill-Nerode theorem for ordered DAG languages, and translate it into a MAT-learning algorithm for the same class. The algorithm infers a minimal OPDG G for the target language in time polynomial in G and the samples provided by the MAT oracle.

1 Introduction

Graphs are one of the fundamental data structures of computer science, and appear in every conceivable application field. We see them as atomic structures in physics, as migration patterns in biology, and as interaction networks in sociology. For computers to process potentially infinite sets of graphs, i.e., graph languages, these must be represented in a finite form akin to grammars or automata. However, the very expressiveness of graph languages often causes problems, and many of the early formalisms have NP-hard membership problems; see, e.g., [15] and [8, Theorem 2.7.1].

Motivated by applications in natural language processing (NLP) that require more light-weight forms of representation, there is an on-going search for grammars that allow polynomial time parsing. A recent addition to this effort was the introduction of order-preserving DAG grammars (OPDGs) [3]. This is a restricted type of hyper-edge replacement grammars [8] that generate languages of directed acyclic graphs in which the nodes are inherently ordered. The authors provide a parsing algorithm that exploits this order, thereby limiting nondeterminism and placing the membership problem for OPDGs in $O\big(n^2 + nm\big)$, where m and n are the sizes of the grammar and the input graph, respectively. This is to be compared with the unrestricted case, in which parsing is NP-complete.

The introduction of OPDGs is a response to the recent application [5] of Hyperedge Replacent Grammars (HRGs) to abstract meaning representations (AMRs) [2]. An AMR is a directed acyclic graph that describes the semantics of a natural language sentence. Although restricted, OPDGs retain enough expressive power to capture AMRs.

In this paper, we continue to explore the OPDGs mathematical properties. We provide an algebraic representation of their domain, and a Myhill-Nerode theorem for the ordered DAG languages. We show that every ordered DAG language L is generated by a minimal unambiguous OPDG G_L, and that this grammar is unique up to renaming of nonterminals. In this context, 'unambiguous' means that every graph is generated by at most one nonterminal. This is similar the behaviour of deterministic automata, in particular that of bottom-up deterministic tree automata which take each input tree to at most one state.

One way of understanding the complexity of the class of ordered DAG languages, is to ask what kind of information is needed to infer its members. MAT learning [1], where MAT is short for minimal adequate teacher, is one of the most popular and well-studied learning paradigms. In this setting, we have access to an oracle (the teacher) that can answer *membership queries* and *equivalence queries*. In a membership query, we present the teacher with a graph g and are told whether g is in the target language L. In an *equivalence* query, we give the teacher an OPDG H and receive in return an element in in the symmetric difference of $L(H)$ and L. This element is called a *counterexample*. If L has been successfully inferred and no counterexample exists, then the teacher instead returns the special token $\perp$.

MAT learning algorithms have been presented for a range of language classes and representational devices [1,16,17,9,11,4,13]. There have also been some results on MAT learning for graph languages. Okada et al. present an algorithm for learning unions of linear graph patterns from queries [14]. These patterns are designed to model structured data (HTML/XML). The linearity of the patterns means that no variable can appear more than once. Hara and Shoudai do MAT learning for context-deterministic regular formal graph systems [10]. Intuitively, the context determinism means that a context uniquely determines a nonterminal, and only graphs derived from this nonterminal may be inserted into the context. Both restrictions are interesting, but neither is compatible with our intended applications.

2 Preliminaries

Sets, sequences and numbers. The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ abbreviates $\{1, \ldots, n\}$, and $\langle n \rangle$ the sequence $1 \cdots n$. In particular, $[0] = \emptyset$ and $\langle 0 \rangle = \lambda$. We also allow the use of sets as predicates: Given a set S and an element s, $S(s)$ is *true* if $s \in S$, and *false* otherwise. When $\equiv$ is an equivalence relation on S, $(S/\equiv)$ denotes the partitioning of S into equivalence classes induced by $\equiv$. The *index* of $\equiv$ is $|(S/\equiv)|$. For $s \in S$, $[s]_\equiv$, or simply $[s]$, is the equivalence class of s with respect to $\equiv$.

Let S° be the set of non-repeating sequences of elements of S. We refer to the ith member of a sequence s as s_i. When there is no risk for confusion, we use sequences directly in set operations, as the set of their members. Given a partial order $\preceq$ on S, the sequence $s_1 \cdots s_k \in S^\circ$ respects $\preceq$ if $s_i \preceq s_j$ implies $i \preceq j$.

A *ranked alphabet* is a pair $(\Sigma, rank)$ consisting of a finite set Σ of symbols and a *ranking function* $rank : \Sigma \mapsto \mathbb{N}$ which assigns a rank $rank(a)$ to every symbol $a \in \Sigma$. The pair $(\Sigma, rank)$ is typically identified with Σ, and the second component is kept implicit.

Graphs. Let Σ be a ranked alphabet. A (directed edge-labelled) *hypergraph* over Σ is a tuple $g = (V, E, src, tar, lab)$ consisting of

- finite sets V and E of *nodes* and *edges*, respectively,
- *source* and *target mappings* $src\colon E \mapsto V$ and $tar\colon E \mapsto V^\circ$ assigning to each edge e its source $src(e)$ and its sequence $tar(e)$ of targets, and
- a *labelling* $lab\colon E \mapsto \Sigma$ such that $rank(lab(e)) = |tar(e)|$ for every $e \in E$.

Since we are only concerned with hypergraphs, we simply call them graphs.

A *path* in g is a finite and possibly empty sequence $p = e_1, e_2, \ldots, e_k$ of edges such that for each $i \in [k-1]$ the source of e_{i+1} is a target of e_i. The *length* of p is k, and p is a *cycle* if $src(e_1)$ appears in $tar(e_k)$. If g does not contain any cycle then it is a *directed acyclic graph* (DAG). The *height* of a DAG G is the maximum length of any path in g. A node v is a *descendant* of a node u if $u = v$ or there is a nonempty path $e_1, \ldots, e_k$ in g such that $u = src(e_1)$ and $v \in tar(e_k)$. An edge e' is a *descendant edge* of an edge e if there is a path $e_1, \ldots, e_k$ in g such that $e_1 = e$ and $e_k = e'$.

The *in-degree* and *out-degree* of a node $u \in V$ is $|\{e \in E \mid u \in tar(e)\}|$ and $|\{e \in E \mid u = src(e)\}|$, respectively. A node with in-degree 0 is a *root* and a node with out-degree 0 is a *leaf*. For a single-rooted graph g, we write $root(g)$ for the unique root node.

For a node u of a DAG $g = (V, E, src, tar, lab)$, the *sub-DAG rooted at* u is the DAG $g{\downarrow}u$ induced by the descendants of u. Thus $g{\downarrow}u = (U, E', src', tar', lab')$ where U is the set of all descendants of u, $E' = \{e \in E \mid src(e) \in U\}$, and src', tar', and lab' are the restrictions of src, tar and lab to E'. A leaf v of $g{\downarrow}u$ is *reentrant* if there exists an edge $e \in E \setminus E'$ such that v occurs in $tar(e)$. Similarly, for an edge e we write $g{\downarrow}e$ for the subgraph induced by $src(e)$, $tar(e)$, and all descendants of nodes in $tar(e)$. This is distinct from $subgg src(e)$ iff $srce$ has out-degree greater than 1.

Marked graphs. Although graphs, as defined above, are the objects we are ultimately interested in, we will mostly discuss marked graphs. When combining smaller graphs into larger ones, whether with a grammar or algebraic operations, the markings are used to know which nodes to merge with which.

A *marked DAG* is a tuple $g = (V, E, src, tar, lab, X)$ where (V, E, src, tar, lab) is a DAG and $X \in V^\circ$ is nonempty. The sequence X is called the *marking* of g, and the nodes in X are referred to as *external nodes*. For $X = v_0 v_1 \cdots v_k$, we write $head(g) = v_0$ and $ext(g) = v_1 \cdots v_k$. We say that two marked graphs are isomorphic modulo markings if their underlying unmarked graphs are isomorphic. The *rank* of a marked graph g is $|ext(g)|$.

Graph operations. Let g be a single-rooted marked DAG with external nodes X and $|ext(g)| = k$. Then g is called a *k-graph* if $head(g)$ is the unique root of g, and all nodes in $ext(g)$ are leaves.

If $head(g)$ has out-degree at most 1 (but is not necessarily the root of g), and either $head(g)$ has out-degree 0 or $ext(g)$ is exactly the reentrant nodes of $g \downarrow head(g)$, then g is a *k-context*. We denote the set of all k-graphs over Σ by $\mathbb{G}^k_\Sigma$, and the set of all k-contexts over Σ by $\mathbb{C}^k_\Sigma$. Furthermore, $\mathbb{G}_\Sigma = \cup_{k\in\mathbb{N}}\mathbb{G}^k_\Sigma$ and $\mathbb{C}_\Sigma = \cup_{k\in\mathbb{N}}\mathbb{C}^k_\Sigma$. Note that the intersection $\mathbb{G}_\Sigma \cap \mathbb{C}_\Sigma$ is typically not empty. Finally, the *empty context* consisting of a single node, which is external, is denoted by ϵ.

Given $g \in \mathbb{G}^k_\Sigma$ and $c \in \mathbb{C}^k_\Sigma$, the *substitution* $c[\![g]\!]$ of g into c is obtained by first taking the disjoint union of g and c, and then merging $head(g)$ and $head(c)$, as well as the sequences $ext(g)$ and $ext(c)$ element-wise. The results is a single-rooted, unmarked DAG.

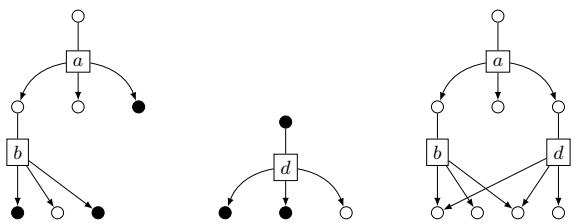

Fig. 1. A 2-context c, a 2-graph g, and the concatenation $c[\![g]\!]$. Filled nodes convey the marking of c and g, respectively. Both targets of edges and external nodes of marked graphs are drawn in order from left to right unless otherwise noted.

Let g be a graph in $\mathbb{G}^0_\Sigma$, e an edge and let h be the marked graph given by taking $g \downarrow e$ and marking the (single) root, and all reentrant nodes. Then the *quotient* of $g \in \mathbb{G}^0_\Sigma$ with respect to h, denoted g/h is the unique context $c \in \mathbb{C}^k_\Sigma$ such that $c[\![h]\!] = g$. The *quotient* of a graph language $L \subseteq \mathbb{G}_\Sigma$ with respect to $g \in \mathbb{G}_\Sigma$ is the set of contexts $L/g = \{c \mid c[\![g]\!] \in L\}$.

Let A be a symbol of rank k. Then $A^\bullet$ is the graph $(V, \{e\}, src, tar, lab, X)$, where $V = \{v_0, v_1, \ldots, v_k\}$, $src(e) = v_0$, $tar(e) = v_1 \ldots v_k$, $lab(e) = A$, and $X = v_0 \ldots v_k$. Similarly, $A^\odot$ is the very same graph, but with only the root marked, in other words, $X = v_0$.

3 Well-ordered DAGs

In this section, we present two formalisms for generating languages of DAGs, one grammatical and one algebraic. Both generate graphs that are *well-ordered* in the sense defined below. We show that the two formalisms define the same families of languages. This allows us to use the algebraic formulation as a basis for the upcoming Myhill-Nerode theorem and MAT learning algorithm.

An edge e with $tar(e) = w$ is a *common ancestor edge* of nodes u and u' if there are t and t' in w such that u is a descendant of t and u' is a descendant of t'. If, in addition, there is no edge with its source in w that is a common ancestor edge of u and u', we say that e is a *closest common ancestor edge* of u and u'. If e is a common ancestor edge of u and v we say that e *orders* u and v, with u before v, if $tar(e)$ can be written as wtw', where t is an ancestor of u and every ancestor of v in $tar(e)$ can be found in w'.

The relation $\preceq_g$ is defined as follows: $u \preceq_g v$ if every closest common ancestor edge e of u and v orders them with u before v. It is a partial order on the leaves of g[3]. Let g be a graph. We call g *well-ordered*, if we can define a total order $\trianglelefteq$ on the leaves of g such that $\preceq_g \subseteq \trianglelefteq$, and for every $v \in V$ and every pair u, u' of leaves of $g{\downarrow}v$, $u \preceq_{g\downarrow v} u'$ implies $u \trianglelefteq u'$.

3.1 Order-preserving DAG grammars

Order-preserving DAG grammars (OPDGs) are essentially hyper-edge replacement grammars with added structural constraints to allow efficient parsing.[1] The idea is to enforce an easily recognisable order on the nodes of the generated graphs, that provides evidence of how they were derived. The constraints are rather strict, but even small relaxations make parsing NP-hard; for details, see [3]. Intuitively, the following holds for any graph g generated by an OPDG:

- g is a connected, single-rooted DAG,
- only leaves of g have in-degree greater than 1, and
- g is well-ordered

Definition 1 (Order-preserving DAG grammar [3]). *An* order-preserving DAG grammar *is a system $H = (\Sigma, N, I, P)$ where Σ and N are disjoint ranked alphabets of* terminals *and* nonterminals*, respectively, I is the set of* starting nonterminals*, and P is a set of* productions*. Each production is of the form $A \to f$ where $A \in N$ and $f \in \mathbb{G}^{rank(A)}_{\Sigma \cup N}$ satisfies one of the following two cases:*

1. *f consists of exactly two nonterminal edges e_1 and e_2, both labelled by A, such that $src(e_1) = src(e_2) = head(f)$ and $tar(e_1) = tar(e_2) = ext(f)$. In this case, we call $A \to f$ a* clone rule*.*
2. *f meets the following restrictions:*
   - *no node has out-degree larger than* 1
   - *if a node has in-degree larger than one, then it is a leaf;*
   - *if a leaf has in-degree exactly one, then it is an external node or its unique incoming edge is terminal*
   - *for every nonterminal edge e in f, all nodes in $tar(e)$ are leaves, and $src(e) \neq head(f)$*
   - *the leaves of f are totally ordered by $\preceq_f$ and $ext(f)$ respects $\preceq_f$.*

[1] In [3], the grammars are called Restricted DAG Grammars, but we prefer to use a name that is more descriptive.

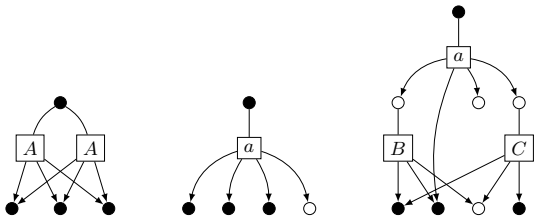

Fig. 2. Examples right-hand sides f of normal form rules of types (a), (b), and (c) for a nonterminal of rank 3.

A derivation step of H is defined as follows. Let $\rho = A \to f$ be a production, g a graph, and g_A a subgraph of g isomorphic modulo markings to $A^{\odot}$. The result of applying ρ to g at g_A is the graph $g' = (g/g_A)[\![f]\!]$, and we write $g \Rightarrow_\rho g'$. Similarly, we write $g \Rightarrow_H^* g'$ if g' can be derived from g in zero or more derivation steps. The language $\mathcal{L}(H)$ of H are all graphs g over the terminal alphabet Σ such that $S^\bullet \Rightarrow_H^* g$, for some $S \in I$. Notice that since a derivation step never removes nodes and never introduces new markings, if we start with a graph g with $|ext(g)| = k$, all derived graphs g' will have $|ext(g')| = k$. In particular, if we start from $S^\bullet$, all derived graphs will have $|ext(g')| = rank(S)$.

Definition 2 (Normal form [3]). *An OPDG H is on* normal form *if every production $A \to f$ is in one of the following forms:*

(a) The rule is a clone rule.
(b) f has a single edge e, which is terminal.
(c) f has height 2, the unique edge e with $src(e) = head(f)$ is terminal, and all other edges are nonterminal.

We say that a pair of grammars H and H' are *language-equivalent* if $\mathcal{L}(H) = \mathcal{L}(H')$. As shown in [3], every OPDG H can be rewritten to a language-equivalent OPDG H' in normal form in polynomial time.

For a given alphabet Σ, we denote the class of graphs $\cup_{H \text{ is an OPDG}} \mathcal{L}(H)$ that can be generated by some OPDG by $\mathcal{H}_\Sigma$, and by $\mathcal{H}_\Sigma^k$ the set of rank k marked graphs that can be generated from a rank k nonterminal.

3.2 DAG concatenation

In sections 4 and 5, we need algebraic operations to assemble and decompose graphs. For this purpose, we define graph concatenation operations that mirror the behaviour of our grammars and show that the class of graphs that can be constructed in this way is equal to $\mathcal{H}_\Sigma$.

In particular, we construct our graphs in two separate ways, mirroring the cloning and non-cloning rules of the grammars:

- 2-concatenation, which takes 2 rank-m graphs and merges their external nodes. This corresponds to the clone rules in Definition 2.

– a-concatenation, for $a \in \Sigma$, takes an a-labelled $rank(a)$ terminal edge and a number (less than or equal to $rank(a)$) of marked graphs, puts the graphs under targets of the terminal edge, and merges some of the leaves. This corresponds to rules of type (b) or (c) in Definition 2.

The second operation is more complex, as we must make sure that the output conforms to the ordering and structural constraints of OPDG. Given a terminal a of rank k and a sequence $g_1, \ldots, g_n$, with $n \leq k$ of marked graphs, we create new graphs in the following way. We start with $a^{\odot}$ and, for each $i \in [n]$ identify $head(g_i)$ with a unique leaf of $a^{\odot}$, intuitively "hanging" $g_1, \ldots, g_n$ under an edge labelled a. We then identify some of the leaves of the resulting graph, but only in such a way that the resulting graph is well-ordered. The intuition is that we mirror productions of type (b) and (c) from Definition 2, but instead of producing a graph containing nonterminal edges, we immediately replace the nonterminals by graphs that can be derived from them. More formally:

Definition 3 (2-concatenation). *Let $m \in \mathbb{N}$ and let $g_1, g_2 \in \mathbb{G}^m_\Sigma$ be disjoint graphs. A* 2-concatenation $2[g_1, g_2]$ *is obtained by merging the roots and external nodes of g_1 and g_2. The root and external nodes of $2[g_1, g_2]$ are the merged roots and external nodes, respectively.*

Observation 4 (k-concatenation) *An obvious extension of* 2*-concatenation is to use an arbitrary number k of graphs from $\mathbb{G}^m_\Sigma$ instead of just* 2*. Such operations can be implemented using iterated* 2*-concatenations, and we refer to them as k-concatenations.*

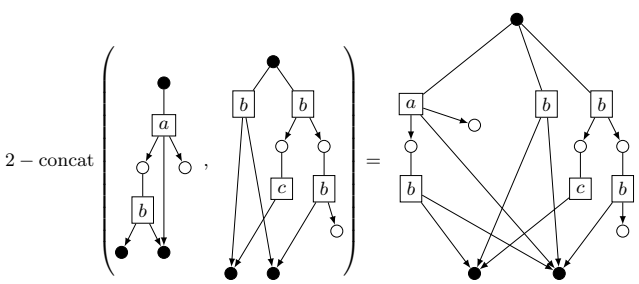

Fig. 3. A 2-concatenation of two graphs

The second operation is more complex, since we must make sure that order is preserved. Given a terminal a of rank k and a sequence $g_1, \ldots, g_n$, with $n \leq k$ of marked graphs, new graphs are created in the following way. We start with $a^{\odot}$ and, for each $i \in [n]$ identify $head(g_i)$ with a unique leaf of $a^{\odot}$, intuitively "hanging" $g_1, \ldots, g_n$ under an edge labelled a. We then identify some of the leaves of the resulting graph. In order to fully specify the result of such a concatenation, and to make sure that it preserves order, we need to parameterize it with the following.

(1) A number m. This is the number of nodes we will merge the external nodes of the graphs $g_1, \ldots, g_n$ and the remaining leaves of the a-labelled edge into.
(2) A subsequence $s = s_1 \ldots s_n$ of $\langle k \rangle$ of length n. This sequence defines under which leaves of $a^{\odot}$ we are going to hang which graph.
(3) A subsequence x of $\langle m \rangle$. This sequence defines which of the leaves of the resulting graph will be external.
(4) An order-preserving function φ that defines which leaves to merge. Its domain consists of the external leaves of the graphs $g_1, \ldots, g_n$ as well as the leaves of $a^{\odot}$ to which no graph from $g_1, \ldots, g_n$ is assigned. Its range is $[m]$.

Before we describe the details of the concatenation operation, we must go into the rather technical definition of what it means for φ to be order-preserving. It has to fulfil the following conditions:

(i) If both u and v are marked leaves of g_i, for some $i \in [n]$, and u comes before v in $ext(g_i)$, then $\varphi(u) < \varphi(v)$.
(ii) If $|\varphi^{-1}(i)| = 1$, then either $i \in x$ or the unique node v with $\varphi(v) = i$ belongs to $a^{\odot}$.
(iii) If there are i and j in $[m]$, with $i < j$ such that no graph g_ℓ for $\ell \in [n]$ contains both a member of $\varphi^{-1}(i)$ and a member of $\varphi^{-1}(j)$, then there exists a $p \in [k]$ such that either
  – p is the qth member of s, and g_q contains a member of $\varphi^{-1}(i)$, or
  – the pth member of $tar(a)$ is in $\varphi^{-1}(i)$
  and furthermore there is no $r < p$ such that either
  – r is the tth member of s and g_t contains a member of $\varphi^{-1}(j)$, or
  – the rth member of $tar(a)$ is itself in $\varphi^{-1}(j)$

Definition 5 (a-concatenation). *Given a terminal a, the a-concatenation of $g_1, \ldots, g_n$, parameterized by m, s, x, ϕ is the graph g obtained by doing the following. For each $i \in [n]$, identify $head(g_i)$ with the leaf of $a^{\odot}$ indicated by s_i. For each $j \in [m]$, identify all nodes in $\varphi^{-1}(j)$. Finally, $ext(g)$ is the subsequence of the m nodes from the previous step indicated by x.*

We note that our concatenation operations can be seen as algebraic operations independent of their input. $2-$concatenation is defined for any pair of graphs, as long as both of them have the same rank. For the $a-$concatenations, things are a little bit more complex, but once we fix the parameters m, s, x, φ, we can see (a, m, s, x, φ) as a well-defined operator that can take any sequence of $|s|$

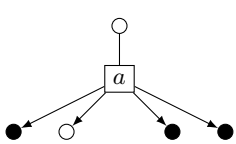

Fig. 4. The topmost terminal graph used in a-concatenation, for a rank-4 terminal symbol a, with the subsequence s of $\langle 4 \rangle$ where subgraphs will be attached indicated as filled leaves

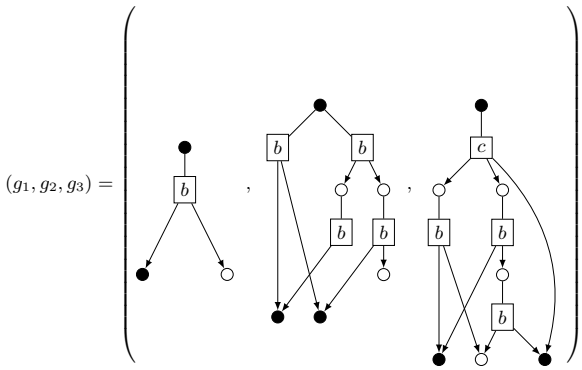

Fig. 5. The (previously constructed) marked graphs, here used as arguments to our example a-concatenation.

Fig. 6. The nodes that leaves are being merged into in an a-concatenation. The subsequence x of external nodes of the concatenated graph are indicated as filled nodes.

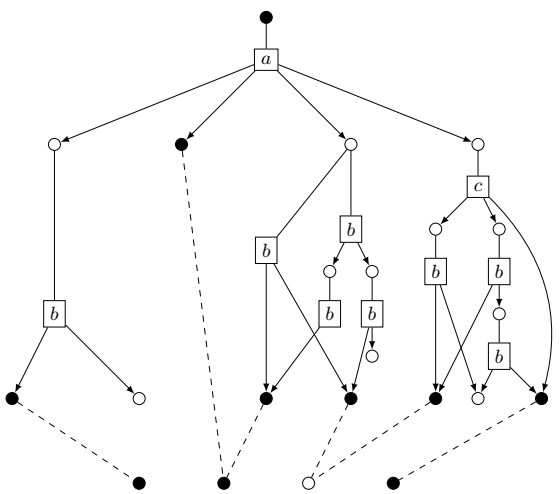

Fig. 7. A "halfway done" a-concatenation, where the input graphs has been hanged underneath the terminal edge, but leaves have not yet been merged. The filled leaves in the graph indicate the domain of φ, and the dashed lines show which node in $\langle m \rangle$ each leaf is merged into. As previously, x is given by the filled nodes of $\langle m \rangle$.

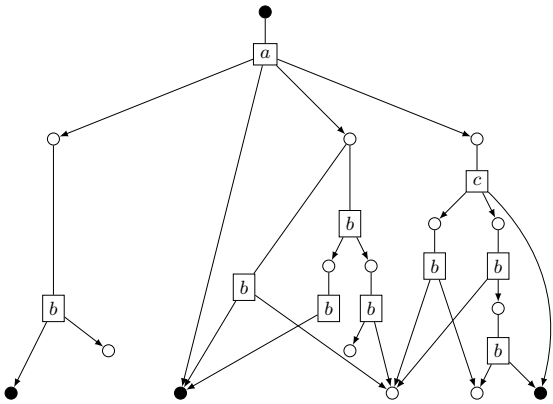

Fig. 8. The marked graphs that is result of the complete a-concatenation, including the merges indicated by φ

graphs as input, as long as their ranks match what φ expects. Indeed, instead of defining the range of φ as the external nodes of the input graphs together with the unused leaves of $a^{\odot}$, i.e., those not indicated by s, we can see it as a function from numbers and pairs of numbers in the following way. If φ is defined for a leaf ℓ of $a^{\odot}$ whose position in $tar(a)$ is i, then we redefine φ so that $\varphi(i) = \varphi(\ell)$. If, on the other hand, ℓ is the jth member of $ext(g_i)$, then we set $\varphi(i, j) = \varphi(\ell)$.

We denote by $\mathcal{A}_\Sigma$ the class of marked graphs that can be assembled from Σ through a- and 2-concatenation, and by $\mathcal{A}^k_\Sigma \subseteq \mathcal{A}_\Sigma$ the graphs of rank k.

Each concatenation operation can be defined as an algebraic operation that takes a number of graphs (of certain ranks) and combines them.

Observation 6 *Let ψ be a concatenation operator and $g_1, \ldots, g_n$ a sequence of graphs for which it is defined. Let $g = \psi(g_1, \ldots, g_n)$. For some $i \in n$, let g' be a graph of the same rank as g_i. Then $\psi(g_1, \ldots, g_{i-1}, g', g_{i+1}, \ldots, g_n) = (g/g_i)[\![g']\!]$.*

The following is the main result of this section.

Theorem 7. *$\mathcal{A}_\Sigma = \mathcal{H}_\Sigma$, and $\mathcal{A}^k_\Sigma = \mathcal{H}^k_\Sigma$ for all k.*

Proof. The proof for $\mathcal{A}_\Sigma \subseteq \mathcal{H}_\Sigma$ is by induction on the size of a graph g. For the base step, we observe that all connected graphs consisting of a single edge with the appropriate number nodes trivially belong to both $\mathcal{A}_\Sigma$ and $\mathcal{H}_\Sigma$. For the inductive step, there are two cases.

We first address 2-concatenation. Let $g = 2[g_1, g_2]$, and assume both g_1 and g_2 are in both $\mathcal{H}^\ell_\Sigma$ and $\mathcal{A}^\ell_\Sigma$, for $\ell = |ext(g)|$. Thus, there are OPDGs H_1 and H_2 generating g_1 and g_2 with some derivations $S^\bullet_i \Rightarrow f_i \Rightarrow^* g_i$ for $i \in \{1, 2\}$. We can construct an OPDG H that generates g by essentially merging H_1 and H_2, and adding a new nonterminal S' of rank ℓ, with a cloning rule and the productions $S' \to f_i$ for $i \in \{1, 2\}$. This will allow the grammar H to generate g, as well as any k-concatenation involving any combination of the same two graphs.

For a-concatenation, we reason as follows. Let g be obtained from $g_1, \ldots, g_n$ by applying a-concatenation, parameterized by m, s, x, φ. Since every g_i is smaller than g, it belongs to $\mathcal{H}_\Sigma$, and hence there is an OPDG H_i with initial nonterminal S_i, such that $S_i^\bullet \Rightarrow^* g_i$. We construct an OPDG H such that $g \in \mathcal{L}(H)$ as follows:

- We add all the productions, nonterminals etc. from H_i, $i \in [n]$, keeping the sets of nonterminals disjoint.
- We add a starting nonterminal S' of rank $|ext(g)|$ and the production $S' \Rightarrow f$ where $f = (a, m, s, x, \varphi)[S_1^\bullet, \ldots, S_n^\bullet]$.

By the context-freeness lemma for HRGs[8], there is a derivation $f \Rightarrow^* g$. It remains to be proved that f is a valid right-hand side, and thus H a valid OPDG. The single edge connected to the root of f is terminal. There are no nodes of out-degree greater than 1, and only a single layer of n nonterminal edges. Leaves of in-degree 1 are either connected to the terminal or are external (or both). This is ensured by item (ii) in the requirements for φ to be order-preserving. Any nonexternal leaves are either connected to the terminal or at least of in-degree 2. Let us now check that the leaves are totally ordered by $\preceq_f$, and moreover that $ext(f)$ respects it. As there are no nodes of out-degree greater than 1, the only way $\preceq_f$ can fail to be total is if two leaves u, v have two closest common ancestor edges e_i, e_j such that u comes before v in $tar(e_i)$, but not in $tar(e_j)$. However, the requirement that φ is order-preserving precludes this.

To prove the opposite direction, let $H = (\Sigma, N, S, P)$ be an OPDG on normal form. We show by induction on the length of the derivations that both the terminal graphs and the intermediate graphs that arise during the derivations are in $\mathcal{A}_{\Sigma \cup N}$.

Again, the base case is trivial — for any start symbol S, the graph $S^\bullet$ clearly belongs $\mathcal{A}_{\Sigma \cup N}$. Moving on to the inductive step, we assume that we have a derivation $S^\bullet \Rightarrow^* g \Rightarrow_{A \to f} g'$ with $g \in \mathcal{A}_{\Sigma \cup N}$. For the rule $A \to f$ to be applicable, g must have a subgraph h that is isomorphic modulo markings to $A^\bullet$. We know that $g' = (g/h)[\![f]\!]$. We first argue that if $f \in \mathcal{A}_{\Sigma \cup N}$, then $g' \in \mathcal{A}_{\Sigma \cup N}$. For since h is a part of g, any construction of g using concatenation operators must at some point use a concatenation operation $\psi(g_1, \ldots, g_n)$, with $g_i = A^\bullet$ for some $i \in [n]$, resulting in a graph h'. By Observation 6, if we use f instead of $A^\bullet$ in this operation, we get a graph $(h'/A^\bullet)[\![f]\!]$. If in all later concatenations, we use this graph instead of h', then, by induction, we will in the end obtain $(g/h)[\![f]\!] = g'$. It remains to show that $f \in \mathcal{A}_{\Sigma \cup N}$. There are three cases:

(1) $A \to f$ is a clone rule, in which case $f = 2[A^\bullet, A^\bullet]$.
(2) f is a single terminal edge, in which case it is also clearly a member of $\mathcal{A}_\Sigma$.
(3) f is of height 2, and the single edge of rank k connected to the root is terminal. This closely mirrors an a-concatenation, where the graphs $g_1, \ldots g_n$ are graphs with just a single nonterminal edge each. The parameters of the concatenation can be read directly from the form of f. The one thing to notice is that the conditions that the leaves of f be totally ordered by $\preceq_f$ and that $ext(f)$ respect $\preceq_f$ ensures that we can find an order-preserving φ and make x be a subsequence of $\langle m \rangle$. □

4 A Myhill-Nerode theorem

We begin by defining the Nerode congruence for ordered DAG languages. From here on, let L be such a language. Intuitively, a pair of graphs are equivalent with respect to L if they can be freely substituted for one another in any context, without disturbing the resulting graph's membership in L. For our purposes, it is useful to view the Nerode congruence as a corner case in a family of relations, each focusing on a subset of $\mathbb{C}_\Sigma$.

Definition 8. *Let $C \subseteq \mathbb{C}_\Sigma$. The equivalence relation $\equiv_{L,C}$ on $\mathcal{H}_\Sigma$ is given by: $g \equiv_{L,C} g'$ if and only if $(L/g \cap C) = (L/g' \cap C)$. The relation $\equiv_{L,\mathbb{C}_\Sigma}$ is known as the* Nerode congruence with respect to L *and is written $\equiv_L$.*

It is easy to see that for two graphs to be equivalent, they must have equally many external nodes. The graph g is *dead* (with respect to L) if $L/g = \emptyset$, and graphs that are not dead are *live*. Thus, if $\equiv_L$ has finite index, there must be a $k \in \mathbb{N}$ such that every $g \in \mathcal{H}_\Sigma$ with more than k external nodes is dead.

In the following, we use $\Psi(\Sigma)$ to denote the set of all concatenation operators applicable to graphs over Σ.

Definition 9 (Σ-expansion). *Given $N \subseteq \mathcal{A}_\Sigma$, we write $\Sigma(N)$ for the set:*

$$\{\psi(g_1, \ldots, g_m) \mid \psi \in \Psi(\Sigma), g_1, \ldots, g_m \in N \text{ and } \psi(g_1, \ldots, g_m) \text{ is defined } \}\,.$$

In the upcoming Section 5, Theorem 13 will form the basis for a MAT learning algorithm. As is common, this algorithm maintains an *observation table* T that collects the information needed to build a finite-state device for the target language L. The construction of an OPDG G^{T} from T is very similar to that from the Nerode congruence, so by introducing it here, we can make use of it twice. Intuitively, the observation table is made up of two sets of graphs N and P, representing nonterminals and production rules, respectively, and a set of contexts C used to explore the congruence classes of $N \cup P$ with respect to L.

To facilitate the design of new MAT learning algorithms, the authors of [6] introduce the notion of an *abstract observation table* (AOT); an abstract data type guaranteed to uphold certain helpful invariants.

Definition 10 (Abstract observation table, see [6]). *Let $N \subseteq P \subseteq \Sigma(N) \subseteq \mathcal{A}_\Sigma$, with N finite. Let $C \subseteq \mathbb{C}_\Sigma$, and let $\rho : P \mapsto N$. The tuple (N, P, C, ρ) is an* abstract observation table with respect to L *if for every $g \in P$,*

1. *$L/g \neq \emptyset$, and*
2. *$\forall g' \in N \setminus \{\rho(g)\} : g \not\equiv_{L,C} g'$.*

The AOT in [6] accommodates production weights taken from general semirings. The version that we have recalled here has three modifications: First, we dispense with the sign-of-life function that maps every graph $g \in N$ to an element in L/g. Its usages in [6] are to avoid dead graphs, and to compute the

weights of productions involving g. From the way new productions and nonterminals are discovered, we already know that they are live, and as we are working in the Boolean setting, there are no transition weights to worry about. Second, we explicitly represent the set of contexts C to prove that the nonterminals in N are distinct. Both realisations of the AOT discussed in [6] collect such contexts, though it is not enforced by the AOT. Third, we do not require that $L(g) = L(\rho(g))$, as this condition is not necessary for correctness, though it may reduce the number of counterexamples needed. The data fields and procedures have also been renamed to reflect the shift from automata to grammars. From here on, we use a bold font when referring to graphs as nonterminals.

Definition 11. *Let $\mathrm{T} = (N, P, C, \rho)$ be an AOT with respect to L. Then G^{T} is the OPDG $(\Sigma, N^{\mathrm{T}}, I^{\mathrm{T}}, P^{\mathrm{T}})$ where $N^{\mathrm{T}} = N$, $I^{\mathrm{T}} = N \cap L$, and*

$$P^{\mathrm{T}} = \{\boldsymbol{\rho(g)} \to \psi(\boldsymbol{\rho(g_1)}, \ldots, \boldsymbol{\rho(g_m)}) \mid g = \psi(g_1, \ldots, g_m) \in P\} \ .$$

In preparation for Theorem 13, we expand our technical vocabulary. Given an ODPG $G = (\Sigma, N, I, P)$ and a nonterminal $\boldsymbol{f} \in N$, $G_{\boldsymbol{f}} = (\Sigma, N, \{\boldsymbol{f}\}, P)$. The grammar G is *unambiguous* if for every $\boldsymbol{g}, \boldsymbol{h} \in N$, $\mathcal{L}(G_{\boldsymbol{g}}) \cap \mathcal{L}(G_{\boldsymbol{h}}) \neq \emptyset$ implies that $\boldsymbol{g} = \boldsymbol{h}$.

Lemma 12. *If $\equiv_L$ has finite index, then there is a $k \in \mathbb{N}_0$ such that for graph $g \in \mathcal{H}_\Sigma$ with more than k external nodes, L/g is empty.*

Proof. If $|ext(g)| \neq |ext(g')|$, then $L/g \cap L/g' = \emptyset$. By Definition 8, this means that either $L/g = L/g' = \emptyset$, or that $g \not\equiv_L g'$. Since $\equiv_L$ has finite index, $\{ext(g) \mid L/g \neq \emptyset\}$ must be bounded from above. □

Theorem 13 (Myhill-Nerode theorem). *The language L can be generated by an OPDG if and only if $\equiv_L$ has finite index. Furthermore, there is a minimal unambiguous OPDG G_L with $\mathcal{L}(G_L) = L$ that has one nonterminal for every live equivalence class of $\equiv_L$. The OPDG G_L is unique up to nonterminal names.*

Proof. We begin by proving the "if" direction. Let $D = \{g \in \mathcal{A}_\Sigma \mid g \text{ is dead}\}$, and N be a selection of representative elements of $(\mathcal{A}_\Sigma / \equiv_L) \setminus \{D\}$. Let $P = \Sigma(N) \setminus D$. Since L has finite index, N and P are finite sets. Finally, let $C = \mathbb{C}_\Sigma$ and, for every $g \in P$, let $\rho(g)$ be the representative of $[g]_{\equiv_L}$ in N. It is easy to verify that $\mathrm{T} = (N, P, C, \rho)$ is an abstract observation table.

Let us now argue by contradiction that (1) $g \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(g)}})$ and $g \notin \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{g'}})$, for every $g \in \mathcal{A}_\Sigma \setminus D$ and $g' \in N \setminus \{\rho g\}$. Suppose that $g \notin \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(g)}})$. We decompose g into $c[\![g']\!]$ such that Statement 1 is not true for $g' = \psi(g_1, \ldots, g_m)$, but it is true for every proper subgraph of g'.

By construction of T, there is a graph $h' = \psi(\rho(g_1), \ldots, \rho(g_m)) \in P$, and hence a production

$$\boldsymbol{\rho(h')} \to \psi(\boldsymbol{\rho(g_1)}, \ldots, \boldsymbol{\rho(g_m)}) \in P^{\mathrm{T}} \ .$$

Since $g_1 \equiv_L \rho(g_1)$, we have $\psi(\rho(g_1), \rho(g_2), \ldots, \rho(g_m) \equiv_L \psi(g_1, \rho(g_2), \ldots, \rho(g_m))$. As $g_i \equiv_L \rho(g_i)$ for all $i \in [m]$, we can repeat the argument $m-1$ times, and learn that

$$h' = \psi(\rho(g_1), \ldots, \rho(g_m)) \equiv_L \psi(g_1, \ldots, g_m) = g' \ .$$

This means that $g' \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(h')}}) = \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(g')}})$, contrary to our initial assumption.

The "if" direction is completed by noticing that since $g \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(g)}})$,

$$g \in \mathcal{L}(G^{\mathrm{T}}) \iff \boldsymbol{\rho(g)} \in I \iff \rho(g) \in L \iff g \in L \ .$$

Now for the proof of the "only if" direction. Assume that L is generated by the OPDG $H = (\Sigma, N, I, P)$. For every $g \in \mathcal{H}_\Sigma$, let $NT(g) = \{A \in N \mid g \in \mathcal{L}(H_A)\}$. We show that if, for $g, g' \in \mathcal{H}_\Sigma$, $NT(g) = NT(g')$, then $L/g = L/g'$. Suppose that $NT(g) = NT(g')$ and that $c \in L/g$. This means that there is a derivation $I \Rightarrow^* c[\![A]\!] \Rightarrow^* c[\![g]\!]$. Since A is also in $NT(g')$, there is an alternative derivation $I \Rightarrow^* c[\![A]\!] \Rightarrow^* c[\![g']\!]$. This is due to the context-freeness of the grammars; see, e.g., [8], and implies that $c \in L/g'$ which proves the claim. As the powerset of N is finite, so is the index of $\equiv_L$. This completes the "only if" direction.

To see that G^{T} is an unambiguous OPDG, we note that if $g, h \in \mathcal{L}(G^{\mathrm{T}})_{\boldsymbol{f}}$ for some $f \in N$, then $g \equiv h$. There cannot be an unambiguous OPDG with fewer nonterminals, since then two graphs belonging to different congruence classes would be generated from the same nonterminal $\boldsymbol{f}$, and since they can *only* be generated from $\boldsymbol{f}$, they would appear in exactly the same set of contexts. G^{T} has thus the minimal number of nonterminals. Neither can any production be removed, as every production is used in the generation of some live graph $g \in P$, and removing it would cancel all graphs on the form $c[\![g]\!]$ from the language. We conclude that G^{T} is a minimal unambiguous OPDG for L, and that it is unique up to renaming of nonterminals. □

Notice that when L only contains ordered ranked trees (i.e., when the root has exactly one child and no node has more than one ancestor), then Theorem 13 turns into the Myhill-Nerode theorem for regular tree languages [12], and the constructed device is essentially the minimal bottom-up tree automaton for L.

5 MAT learnability

In Section 4, the data fields N, P, and C of the AOT were populated with what is usually called a *characteristic set* for L, to derive the minimal unambiguous OPDG G_L that generates L. In this section, we describe how the necessary information can be incrementally built up by querying a MAT oracle. The learning algorithm interacts with the oracle through the following procedures:

- Equals?(H) returns a graph in $\mathcal{L}(H) \ominus L = \{g \mid \mathcal{L}(H)(g) \neq L(g)\}$, or $\perp$ if no such exists.
- Member?(g) returns the Boolean value $L(g)$.

The information gathered from the oracle is written and read from the AOT through the procedures listed below. In the declaration of these, (N, P, C, ρ) and (N', P, C', ρ') are the data values before and after application, respectively. The procedures are then as follows:

- INITIALISE sets $N' = P' = C' = \emptyset$.
- ADDPRODUCTION(g) with $g \in \Sigma(N) \setminus P$. Requires that $L/g \neq \emptyset$, and guarantees that $N \subseteq N'$ and $P \cup \{g\} \subseteq P'$.
- ADDNONTERMINAL(c, g) with $g \in P \setminus N$ and $c \in \mathbb{C}_\Sigma$. Requires that $\forall g' \in N : g \not\equiv_{L, C \cup \{c\}} g'$, and guarantees that $N \cup \{g\} \subseteq N'$, $P \subseteq P'$, and $C \subseteq C' \subseteq C \cup \{c\}$.
- GRAMMAR returns G^{T} without modifying the data fields.

Algorithms 1 and 2 are recalled almost exactly as they stand in [6], with the only adjustments being those needed to go from weighted automata to unweighted grammars. Algorithm 1 maintains an AOT T, from which it induces an OPDG G^{T}. This OPDG is given to the language oracle LANG in the form of an equivalence query. If the oracle responds with the token $\perp$, then the language has been successfully acquired. Otherwise, the algorithm receives a counterexample $g \in \mathcal{L}(G^{\mathrm{T}}) \ominus L$, from which it extracts new facts about L through the procedure EXTEND and includes these in T.

The technique used in Algorithm 2, EXTEND, is known as *contradiction backtracking*. We cover it superficially here; a closer discussion is available in [7]. The contradiction backtracking essentially consists of simulating the parsing of the counterexample g with respect to the OPDG G^{T}. The simulation is done incrementally, and in each step a subgraph $h \in \Sigma(N) \setminus N$ of g is nondeterministically selected. If h is not in P, this indicates that a production is missing from G^{T} and the problem is solved by a call to ADDPRODUCTION. If h is in P, then the algorithm replaces it by $\rho(h)$ and checks whether the resulting graph g' is in L. If its membership has changed (i.e., if $L(g) \neq L(g')$), then

evidence has been found that h and $\rho(h)$ do not represent the same congruence class and the algorithm calls ADDNONTERMINAL. If the membership has not changed, then the procedure calls itself recursively with the graph g' as argument, which has strictly fewer subgraphs not in P. Since g is a counterexample, so is g'.

If this parsing process succeeds in replacing all of g with a graph $g' \in N$, then $L(g) = L(g')$ and $g \in \mathcal{L}(G^{\mathrm{T}}_{g'})$. Since $g' \in N$, $\mathcal{L}(G^{\mathrm{T}})(g') = L(g')$. It follows that $\mathcal{L}(G^{\mathrm{T}})(g) = L(g)$ which contradicts g being a counterexample.

From [6], we know that if EXTEND adheres to the pre- and postconditions of the AOT procedures, and the target language L can be computed by an OPDG, then Algorithm 1 terminates and returns a minimal OPDG generating L. It thus remains to add realisations of ADDPRODUCTION and ADDNONTERMINAL, and to show that all procedures behave as desired.

Algorithm 1: Template learning algorithm [6]

```
T.Initialise();
while true do
    G^T ← T.Grammar();
    g ← Lang.Equal?(G^T);
    if g = ⊥ then
        return G^T
    else
        T.Extend(g)
```

Algorithm 2: The procedure Extend [6]

```
Data: g ∈ L(G^T) ⊖ L
Decompose g into g = c⟦h⟧ where c ∈ ℂ_Σ, h ∈ Σ(N) \ N;
if h ∉ P then
    T.AddProduction(h);
else
    if Lang.Member?(c⟦h⟧) ≠ Lang.Member?(c⟦ρ(h)⟧) then
        T.AddNonterminal(c, h);
    else
        Extend(c⟦ρ(h)⟧);
```

Consider the implementations of AddProduction and AddNonterminal, shown in Algorithm 3 and Algorithm 4, respectively. AddProduction simply adds its argument g to the set P of graphs representing productions. It then looks for a representative g' for g in N, such that $g' \equiv_{L,C} g$. If no such graph exists, it simply chooses any $g' \in N$, or if N is empty, adds g itself to N with a call to AddNonterminal. Similarly, AddNonterminal adds g to the set N of graphs representing nonterminals. If g cannot be distinguished from $\rho(g)$, which is the only element in N that could possibly be indistinguishable from g, then c is added to C to tell g and $\rho(g)$ apart. Finally, the representative function ρ is updated to satisfy Condition 2 of Definition 10.

Algorithm 3: The procedure AddProduction

```
Data: p ∈ Σ(N) \ P
P ← P ∪ {g};
if ∃g' ∈ N : g ≡_{L,C} g' then
    ρ(g) ← g';
else
    if ∃g' ∈ N then
        ρ(g) ← g';
    else
        AddNonterminal(ϵ, g);
```

Algorithm 4: The procedure AddNonterminal

Data: $g \in P \setminus N$, $c \in \mathbb{C}_\Sigma$, and $\forall g' \in N : g \not\equiv_{L,C\cup\{c\}} g'$
$N \leftarrow N \cup \{g\}$;
if $g \equiv_{L,C} \rho(g)$ **then**
  $C \leftarrow C \cup \{c\}$;
$g' \leftarrow \rho(g)$;
for $h \in \rho^{-1}(g')$ **do**
  if $h \equiv_{L,C} g$ **then**
    $\rho(h) \leftarrow g$;

Lemma 14. *For every $g \in P$, $g \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{\rho(g)}})$.*

Proof. We first prove that for every $g \in N$, $g \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{g}})$. The argument is by induction on the number of edges in g. If g consists of a single edge, then $g = \psi$ for some concatenation operator of rank 0, so the result is trivially true. Assume now that $g = \psi(g_1, \ldots, g_m)$. Since $N \subseteq P \subseteq \Sigma(N)$, there is a production $\boldsymbol{g} \to \psi(\boldsymbol{g_1}, \ldots, \boldsymbol{g_m}) \in P^{\mathrm{T}}$ and $g_i \in N$, $i \in [m]$. By the induction hypothesis, $g_i \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{g_i}})$, $i \in [m]$. It follows that $g \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{g}})$.

Assume now that $g = \psi(g_1, \ldots, g_m) \in P$. Since $P \subseteq \Sigma(N)$, $g_i \in N$, $i \in [m]$. By the above argument, $g_i \in \mathcal{L}(G^{\mathrm{T}}_{\boldsymbol{g_i}})$ for every $i \in [m]$, and since $g \in P$,

$$\boldsymbol{\rho(g)} \to \psi(\boldsymbol{\rho(g_1)}, \ldots, \boldsymbol{\rho(g_m)}) \;=\; \boldsymbol{\rho(g)} \to \psi(\boldsymbol{g_1}, \ldots, \boldsymbol{g_m}) \in P^{\mathrm{T}}$$

so $g \in \mathcal{L}(G^{\mathrm{T}})_{\boldsymbol{\rho(g)}}$. □

Theorem 15. *Algorithm 1 terminates and returns G_L.*

Proof. It should be clear that Initialise trivially fulfils the conditions of Definition 10, and that Grammar has no effect on the data fields at all. Since AddProduction depends on AddNonterminal, we begin verifying the latter.

We us assume that $g \in P \setminus N$, $c \in \mathbb{C}_\Sigma$, and that $\forall g' \in N : g' \not\equiv_{L,C\cup\{c\}} g$. Since N is updated to $N \cup \{g\}$, and P is unchanged, the guarantees of AddNonterminal are fulfilled. Condition 1 of Definition 10 is not affected, and the requirement that $\forall g' \in N : g' \not\equiv_{L,C\cup\{c\}}$ ensures that Condition 2 continues to hold.

Let us now look at AddProduction. Here, we assume that $p \in \Sigma(N) \setminus P$, $c \in \mathbb{C}_\Sigma$, and $L/g \neq \emptyset$, which immediately fulfils Condition 1 of Definition 10, and since N is not updated, Condition 2 is trivially met. Finally we note that in the call to AddNonterminal, $N = \emptyset$, so ϵ trivially a separates g from every other graph in N.

We conclude by ensuring that the AddProduction and AddNonterminal are called from Extend with their requirements met. In case of AddProduction, we know that $c[\![g]\!] \in L$ since $g \notin P$ so $c[\![g]\!] \notin \mathcal{L}(G^{\mathrm{T}})$ and $c[\![g]\!]$ is supposed to be a counterexample. This means in particular that $\{c\} \subseteq L/g$, so L/g is not empty. Also the requirement of AddProduction is met due to the if-clause

on Line 2, since by assumption $\forall g' \in N \setminus \{\rho(g)\} : g' \not\equiv_{L,C} g$ and we know that $c \in L/g \ominus L/\rho(g)$.

Since the Conditions of Definition 10 are respected, and the associated procedures have their requirements met and fulfil their guarantees, [6, Corollary 8] ensures that the learning algorithm terminates and outputs G_L. □

We close this section with a discussion of the complexity of Algorithm 1. To infer the minimal unambiguous ODGP $G_L = (\Sigma, N, I, P)$ recognising L, the algorithm must gather as many graphs as there are nonterminals and transitions in G_L. In each iteration of the main loop, it parses a counterexample g in polynomial time in the size of g and T (the latter is limited by the size of G_L), and is rewarded with at least one production or nonterminal. The algorithm is thus polynomial in $|G_L| = |N| + |P|$ and the combined size of the counterexamples provided by the MAT oracle.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
2. L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. Abstract meaning representation for sembanking. In *7th Linguistic Annotation Workshop (ACL 2013 Workshop)*, 2013.
3. H. Björklund, F. Drewes, and P. Ericson. Between a rock and a hard place – uniform parsing for hyperedge replacement DAG grammars. In A.-H. Dediu, J. Janousek, C. Martín-Vide, and B. Truthe, editors, *10th International Conference on Language and Automata Theory and Applications, Prague, Czech Republic, 2016*, volume 9618 of *Lecture Notes in Computer Science*, pages 521–532. Springer, 2016.
4. J. Björklund, H. Fernau, and A. Kasprzik. Polynomial inference of universal automata from membership and equivalence queries. *Information and Computation*, 246:3–19, 2016.
5. D. Chiang, J. Andreas, D. Bauer, K. M. Hermann, B. Jones, and K. Knight. Parsing graphs with hyperedge replacement grammars. In *51st Annual Meeting of the Association for Computational Linguistics (ACL 2013)*, volume Volume 1: Long Papers, pages 924–932. The Association for Computer Linguistics, 2013.
6. F. Drewes, J. Björklund, and A. Maletti. MAT learners for tree series: an abstract data type and two realizations. *Acta Informatica*, 48(3):165, 2011.
7. F. Drewes and J. Högberg. Query learning of regular tree languages: How to avoid dead states. *Theory of Computing Systems*, 40(2):163–185, 2007.
8. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars*, volume 1, pages 95–162. World Scientific, 1997.
9. F. Drewes and H. Vogler. Learning deterministically recognizable tree series. *Journal of Automata, Languages and Combinatorics*, 12(3):332–354, 2007.
10. S. Hara and T. Shoudai. Polynomial time MAT learning of c-deterministic regular formal graph systems. In *International Conference on Advanced Applied Informatics (IIAI AAI 2014)*, pages 204–211, 2014.
11. J. Högberg. A randomised inference algorithm for regular tree languages. *Natural Language Engineering*, 17(02):203–219, 2011.
12. Dexter Kozen. On the Myhill-Nerode theorem for trees. *Bulletin of the EATCS*, 47:170–173, 1992.
13. A. Maletti. Learning deterministically recognizable tree series—revisited. In *Algebraic Informatics*, pages 218–235. Springer, 2007.
14. R. Okada, S. Matsumoto, T. Uchida, Y. Suzuki, and T. Shoudai. Exact learning of finite unions of graph patterns from queries. In *The 18th International Conference on Algorithmic Learning Theory (ALT 2007)*, pages 298–312, 2007.
15. G. Rozenberg and E. Welzl. Boundary NLC graph grammars—basic definitions, normal forms, and complexity. *Information and Control*, 69(1-3):136–167, 1986.
16. Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2–3):223–242, 1990.
17. H. Shirakawa and T. Yokomori. Polynomial-time MAT learning of c-deterministic context-free grammars. *Transaction of Information Processing Society of Japan*, 34:380–390, 1993.

III

Minimisation and Characterisation of Order-Preserving DAG Grammars

Henrik Björklund[a], Johanna Björklund[a], Petter Ericson[a]

[a]*Department of Computing Science, Umeå University, Sweden*

Order-preserving DAG grammars (OPDGs) is a formalism for processing semantic information in natural languages [5, 4]. OPDGs are sufficiently expressive to model abstract meaning representations, a graph-based form of semantic representation in which nodes encode objects and edges relations. At the same time, they allow for efficient parsing in the uniform setting, where both the grammar and subject graph are taken as part of the input.

In this article, we introduce an initial algebra semantic for OPDGs, which allows us to view them as regular tree grammars. This makes it possible to transfer a number of results from that domain to OPDGs, both in the unweighted and the weighted case. In particular, we show that deterministic OPDGs can be minimised efficiently, and that they are learnable in the so-called MAT setting. To conclude, we show that the languages generated by OPDGs are MSO-definable.

1. Introduction

Order-Preserving DAG Grammars (OPDGs) [5] is a subclass of Hyper-Edge Replacement Grammars (HRGs) [12], motivated by the need to model semantic information in natural-language processing. In OPDGs, the basic units of computation are *directed hyperedges*, the generalisation of regular directed edges that comes from permitting any finite number of target vertices. The left-hand side of a production rule is a single k-targeted hyperedge labelled by a nonterminal symbol, and the right-hand side is a graph with $k+1$ marked vertices. The generation process starts out from an initial graph in which the edges are labelled with nonterminals or terminals. It then iteratively replaces nonterminal edges by larger graph fragments, until only terminal edges remain. The replacement step involves a simple form of graph concatenation, illustrated in Figure 1.

To ensure efficient parsing, the graphs that appear as right-hand sides in OPDG productions must be on one of three allowed forms, illustrated in Figure 2. As a result, the generated graphs are acyclic, rooted, and have a natural order on their nodes. This is restrictive compared to HRGs in general, but sufficiently expressive to model semantic representations such as abstract meaning representations [2]. Moreover, this normal form places parsing in $O(n^2 + nm)$, where m and n are the sizes of the grammar and the input graph, respectively. For full HRGs, parsing is NP-complete even in the non-uniform case, when the grammar is fixed and only the graph is considered as input; see, for example, [12]. In [5], it is shown that even small relaxations of the restrictions on the right-hand sides lead to NP-complete parsing as well.

In [4], we provided an algebraic representation of the languages generated by OPDGs. This allowed us to state and prove a Myhill-Nerode theorem for order-preserving DAG grammars, and in doing so also provide a canonical form and an Angluin-style MAT learning algorithm. In the present work, we generalise these results to the weighted case. This is done by providing an initial algebra semantic for OPDGs, which allows us to transfer a number

of results from the tree case. We also introduce the notion of bottom-up determinism for OPDGs and provide an efficient minimisation algorithm for weighted OPDGs.

A further area of study regarding graph grammars is their relation to logic. In particular, the relation between Monadic Second-Order (MSO) logic on graphs and HRG is an active research topic, with recent results [14] exploring Regular Graph Grammars, a formalism that is both a subclass of HRL and MSO definable. We show that OPDGs occupy a similar position by proving that for every OPDG, we can construct an MSO formula that defines the same graph language.

Both the regular graph grammars of Gilroy et al. [14] and the grammars proposed by Chiang et al. [10] are variants of HRGs that are potential candidates for modelling natural language semantic data. Unlike OPDGs, however, none of these models allow for polynomial time parsing. Further related work include efforts on the generalisation of OPDGs to cover restricted types of cyclic graphs [6]. Additionally, the Regular DAG Automata proposed by Chiang et al. [11] is a recent graph formalism, also studied in, e.g., [8, 3], intended for the same applications as the present work. It shares some desirable properties with OPDGs, though not polynomial time parsing.

2. Preliminaries

Sets, sequences and numbers. The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ abbreviates $\{1, \ldots, n\}$. In particular, $[0] = \emptyset$. We also allow the use of sets as predicates: Given a set S and an element s, $S(s)$ is *true* if $s \in S$, and *false* otherwise. When $\equiv$ is an equivalence relation on S, $(S/\equiv)$ denotes the partitioning of S into equivalence classes induced by $\equiv$. For $s \in S$, $[s]_\equiv$ is the equivalence class of s with respect to $\equiv$.

Let S and T be sets. The set of all bijective functions from S to T is denoted $\mathrm{biject}(S, T)$. Note that $\mathrm{biject}(S, T) = \emptyset$ unless $|S| = |T|$.

Let $S^{\circledast}$ be the set of non-repeating sequences of elements of S. We refer to the ith member of a sequence s as s_i. Given a sequence s, we write $[s]$ for the set of elements of s. Given a partial order $\preceq$ on S, the sequence $s_1 \cdots s_k \in S^{\circledast}$ respects $\preceq$ if $s_i \preceq s_j$ implies $i \leq j$. We write $S^{\oplus}$ for $S^{\circledast} \setminus \{\lambda\}$ where λ denotes the empty sequence.

Ranked alphabets and trees. A *ranked alphabet* is a pair (Σ, rk) consisting of a finite set Σ of symbols and a *ranking function* $rk : \Sigma \to \mathbb{N}$ which assigns a rank $rk(a)$ to every $a \in \Sigma$. The pair (Σ, rk) is typically identified with Σ, and the second component is kept implicit.

The set $\mathbb{T}_\Sigma$ of *trees* over the ranked alphabet Σ is defined inductively as follows:

- Every symbol $f \in \Sigma$ of rank 0 is a tree.
- Every *top-concatenation* $f[t_1, \ldots, t_k]$ of a symbol $f \in \Sigma$ of rank k with trees $t_1 \ldots t_k \in \mathbb{T}_\Sigma$ is a tree.

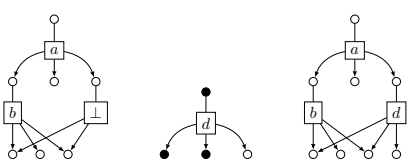

Figure 1: A graph context c, a graph g, and the substitution of g into c. Filled nodes indicate the marking of g.

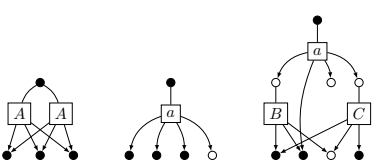

Figure 2: Example right-hand sides.

From here on, let X be a ranked alphabet containing only 0-ranked symbols, called variables, disjoint from every other alphabet discussed here. The set $\mathbb{T}_\Sigma(X)$ is the set of trees over $\Sigma \cup X$. A *tree language* is a subset of $\mathbb{T}_\Sigma$.

A *context over* Σ is a tree in $\mathbb{T}_\Sigma(X)$ containing exactly one occurrence of a symbol in X. The set of contexts over Σ is written $\mathbb{C}_\Sigma$. The substitution of $t \in \mathbb{T}_\Sigma$ into $c \in \mathbb{C}_\Sigma(X)$ is $c[\![t]\!] = c[x \leftarrow t]$ for x the single symbol from X. The tree t is a *subtree* of $s \in \mathbb{T}_\Sigma$ if there is a $c \in \mathbb{C}_\Sigma$, such that $s = c[\![t]\!]$. If t is a tree and v a position in t, we write t/v for the subtree of t rooted at v.

Typed alphabets and graphs. A *typed ranked alphabet* is a tuple (Σ, rk, tp), where (Σ, rk) is a ranked alphabet, and $tp : \Sigma \to \mathbb{N} \times \mathbb{N}^*$ assigns a type $tp(a) \in \mathbb{N} \times \mathbb{N}^{rk(a)}$ to every symbol $a \in \Sigma$. For $tp(a) = (o, i)$, where $o \in \mathbb{N}$ and $i \in \mathbb{N}^*$, we call o the *output type* and i the sequence of *argument types*, respectively, and write $otp(a) = o$, $atp(a) = i$.

Definition 2.1 (hypergraph)**.** *A directed, edge-labeled, marked* hypergraph *over a ranked alphabet* Σ *is a tuple* $g = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ *with the following components:*

- V *and* E *are disjoint finite sets of* nodes *and* edges*, respectively.*
- *The* attachment $\mathrm{att} : E \to V^\oplus$ *assigns a sequence of nodes to each hyperedge. For* $\mathrm{att}(e) = vw$ *with* $v \in V$ *and* $w \in V^\circledast$*, we call* v *the* source *and* w *the sequence of* targets*, respectively, and write* $\mathrm{src}(e) = v$ *and* $\mathrm{tar}(e) = w$*.*
- *The* labeling $\mathrm{lab}\colon E \to \Sigma$ *assigns a label to each edge, subject to the condition that* $\mathrm{rank}(\mathrm{lab}(e)) = |\mathrm{tar}(e)|$ *for every* $e \in E$*.*
- *The sequence* $\mathrm{ext} \in V^\oplus$ *is the sequence of* external nodes*. If* $\mathrm{ext}_G = vw$*, then the node* v *is denoted by* $\dot{g}$ *and the sequence* w *of nodes by* $g_{..}$*, respectively, and we impose the additional requirement that* $\mathrm{src}(e) \notin [g_{..}]$ *for all* $e \in E$*. The type* $tp(g)$ *of* g *is* $(|g_{..}|, \varepsilon)$*.*

In the following, we will only deal with the directed, edge-labeled, marked hypergraphs from Definition 2.1, and will therefore simply call them *graphs*.

A *path* in g is a finite and possibly empty sequence $\rho = e_1 e_2 \cdots e_k$ of edges such that for each $i \in [k-1]$ the source of e_{i+1} is a target of e_i. The *length* of ρ is k, and ρ is a *cycle* if $\mathrm{src}(e_1)$ appears in $\mathrm{tar}(e_k)$. If g does not contain any cycle then it is a *directed acyclic graph* (DAG). The *height* of a DAG G is the maximum length of any path in g. A node v is a *descendant* of a node u if $u = v$ or there is a nonempty path $e_1 \cdots e_k$ in g such that $u = \mathrm{src}(e_1)$ and $v \in [\mathrm{tar}(e_k)]$. An edge e' is a *descendant edge* of an edge e if there is a path $e_1 \cdots e_k$ in g such that $e_1 = e$ and $e_k = e'$. An edge or node is an *ancestor* of its descendants. The *in-degree* and *out-degree* of a node $u \in V$ is $|\{e \in E \mid u \in [\mathrm{tar}(e)]\}|$ and $|\{e \in E \mid u = \mathrm{src}(e)\}|$, respectively. A node with in-degree 0 is a *root* and a node with out-degree 0 is a *leaf*. For a single-rooted graph g, we write $\mathrm{root}(g)$ for the root node.

If A is a nonterminal of rank k, we write $A^\bullet$ for the graph consisting of a single edge, labeled A, with its $k+1$ attached nodes, which are all external.

For nodes u and v of a DAG $g = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$, a node or edge x is a *common ancestor* of u and v if it is an ancestor of both. It is a *closest common ancestor* if there is no descendant of x that is a common ancestor of u and v. A closest common ancestor edge e orders u before v if e's ith target is an ancestor of u, and for all j such that e's jth target is an ancestor of v, $i < j$. The partial order $\preceq_g$ on the leaves of a graph g is, if defined, the reflexive and transitive closure of the relation $\mathrm{before}(u, v)$, which holds if u, v have at least one closest common ancestor edge and if all such edges order u before v.

For a node u of a marked DAG $g = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$, the *sub-DAG rooted at* u is the DAG $g{\downarrow_u}$ induced by the descendants of u. Thus $g{\downarrow_u} = (U, E', \mathrm{att}', lab', \mathrm{ext}')$ where U is

the set of all descendant nodes of u, $E' = \{e \in E \mid \mathrm{src}(e) \in U\}$, and att', and lab' are the restrictions of att and lab to E'. A leaf v of $g{\downarrow_u}$ is *reentrant* in regards to u if there exists an edge $e \in E \setminus E'$ such that v occurs in $\mathrm{tar}(e)$ or in ext. We define ext' to be the sequence starting with u, and continuing with the reentrant nodes of u, ordered by $\preceq_g$, if defined. If $\preceq_g$ is not defined, we let ext' consist only of u. We note that $\preceq_{g{\downarrow_u}}$ is defined and is a subset of $\preceq_g$, if $\preceq_g$ is defined. We also note that if $y \in g{\downarrow_x} \setminus \mathrm{ext}_{g{\downarrow_x}}$ then $g{\downarrow_x}{\downarrow_y} = g{\downarrow_y}$. For proofs of these properties, see [5, 4]. For both nodes and edges x, the set of reentrant leaves of x in the graph g is denoted $\mathrm{reent}_g(x)$.

For an edge e we write $g{\downarrow_e}$ for the subgraph induced by $\mathrm{src}(e)$, $\mathrm{tar}(e)$, and all descendants of nodes in $\mathrm{tar}(e)$, with the same reasoning as above on the definition of ext'. This is distinct from $g{\downarrow_{\mathrm{src}(e)}}$ if and only if $\mathrm{src}(e)$ has out-degree greater than 1.

Let $g = (V_g, E_g, \mathrm{att}_g, \mathrm{lab}_g, \mathrm{ext}_g)$ and $h = (V_h, E_h, \mathrm{att}_h, \mathrm{lab}_h, \mathrm{ext}_h)$ be DAGs. We say that g and h are *isomorphic*, and write $g \approx h$, if there are two bijective functions $f_V : V_g \to V_h$ and $f_E : E_g \to E_h$ such that $\mathrm{att}_h \circ f_E = f_V \circ \mathrm{att}_g$, $\mathrm{lab}_h \circ f_E = \mathrm{lab}_g$, and $\mathrm{ext}_H = f_V(\mathrm{ext}_G)$.

For graphs g, h, f and an edge $e \in E_h$ with $|\mathrm{tar}_h(e)| = |f_{\bullet\bullet}|$, we call $g = h[\![e : f]\!]$ the *graph substitution* of e by f in h, if

- $E_g = E_h \setminus \{e\} \cup E_f$
- $V_g = V_h \cup V_f$
- $\mathrm{ext}_g = \mathrm{ext}_h$

and $\mathrm{att}_g(e') = \mathrm{att}_f(e'), \mathrm{lab}_g(e') = \mathrm{lab}_f(e')$ for $e' \in E_f$, and $\mathrm{att}_g(e') = \mathrm{att}_h(e'), \mathrm{lab}_g(e') = \mathrm{lab}_h(e')$ for $e' \in E_h \setminus \{e\}$. We require that $\mathrm{att}_h(e) = \mathrm{ext}_f = V_f \cap V_h$. Note that we can always choose isomorphic copies of f and h such that this is the case.

For $e, e' \in E_h$, $g = h[\![e : f]\!]$ and $g' = g[\![e' : f']\!]$, we write $g' = h[\![e : f, e' : f']\!]$, and extend this notation to any number of edges in h.

3. Well-ordered DAGs

In this section, we define a universe of *well-ordered DAGs* and discuss formalisms for expressing subsets of this universe, i.e., well-ordered DAG languages (WODLs).

Well-ordered DAGs were initially introduced as the class of graphs recognised by *order-preserving DAG grammars*[1] (OPDG) [5]. Some further properties of OPDGs are studied in [4]. Intuitively, every DAG generated by an ODPG has a partial order on its node set. This order is easily decidable from the structure of the DAG, and simplifies several processing tasks, most notably parsing.

3.1. Order-preserving DAG algebras

Well-ordered DAGs can be inductively assembled using *concatenation operations*, analogously to the step-wise construction of strings or trees through the concatenation of symbols from an alphabet. In the string case, each symbol is a string, and concatenating a string with a symbol yields a new string. In the tree case, each rank-0 symbol is a tree, and top concatenating k trees with a rank-k symbol yields a new tree.

In our domain of well-ordered DAGs, every concatenation operation is assigned a *type* that reflects the structure of the graphs it takes as input and the graph it produces as output. The operations are based on *concatenation schemata*, which also have types. Concatenation schemata are special kinds of DAGs, where some edges are *place-holders* and carry no label.

[1] In [5], the grammars were called "restricted DAG grammars", but in [4], the more descriptive name "order-preserving DAG grammars" was substituted.

Definition 3.1. *Let Σ be a ranked alphabet. A DAG f is a* concatenation schema *over Σ if either of the two following conditions hold.*

1. *f contains exactly two edges, both of rank k, both place-holders, and both have the same source and the same targets, in the same order. All nodes of f are external and connected to the two edges. We call such a graph a* clone. *Its type is (k, kk).*
2. *f has height at most two and satisfies the following.*
   - *No node has an out-degree larger than one.*
   - *There is a single root with a single edge attached to it. This edge is labeled by a terminal from Σ.*
   - *All other edges are place-holders.*
   - *Only leaves have in-degrees larger than one.*
   - *All targets of place-holder edges have in-degree larger than one or are external.*
   - *The ordering $\preceq_f$ is total on the leaves and is respected by $f_{\bullet\bullet}$.*

For a concatenation schema that is not a clone, there is a natural ordering on the place-holder edges. This is because there is a unique edge connected to the root, all place-holders have targets of this edge as sources, and no two place-holders share a source. Thus, if f has ℓ place-holders, we can refer to them as $f_1, \ldots, f_\ell$. In the case of clone rules, the two edges are isomorphic, and we can simply pick any ordering. The number ℓ of place-holder edges in f is the *arity* of f, denoted $\mathit{arity}(f)$. The type of such concatenation schema is $(|f_{\bullet\bullet}|, |\mathrm{tar}_f(f_1)||\mathrm{tar}_f(f_2)|\cdots|\mathrm{tar}_f(f_l)|)$.

Each concatenation schema f gives rise to a concatenation operator concat_f of arity $\mathit{arity}(f)$ as described in the following definition.

Definition 3.2. *Let f be a concatenation schema of type $(o, a_1 \ldots a_\ell)$, and $f_1 \ldots f_\ell$ its place-holder edges. The concatenation operation $\mathrm{concat}_f(g_1, \ldots, g_\ell)$ is defined for well-ordered DAGs $g_1 \ldots g_\ell$ where $\mathit{otp}(g_i) = a_i$ for all $i \in [\ell]$. It yields the graph $g = f[\![f_1 : g_1, \ldots, f_\ell : g_\ell]\!]$.*

If f is a concatenation schema over Σ, we call concat_f a *concatenation operator* over Σ. The set of all such operators is denoted concat_Σ.

A special case of concatenation schemata is the one where the graph f has height one, but is not a clone. In this case, f consists of a single terminal edge. The external nodes include the source and any subsequence of the targets.

Definition 3.3. *Let Σ be an alphabet. The* well-ordered *DAGs over Σ, denoted $\mathcal{A}_\Sigma$, is the set of graphs that can be constructed using operations from* concat_Σ.

3.2. Order-preserving DAG grammars

Order-preserving DAG grammars (OPDGs) produce well-ordered DAGs [5, 4]. In other words, every language produced by an OPDG over Σ is a subset of $\mathcal{A}_\Sigma$. When we next recall the definition, we restrict ourselves to grammars on a particular normal form. As shown in [5], every OPDG can be rewritten into one on this normal form in polynomial time.

If Σ is a ranked alphabet of terminals and N a ranked alphabet of non-terminals, we call a graph f an *N-instantiated concatenation schema over Σ* if f can be obtained from a concatenation schema over Σ by assigning each place-holder a nonterminal from N of appropriate rank.

An order-preserving DAG grammar (OPDG) is a structure $G = (\Sigma, N, P, S)$ where

- Σ is the ranked alphabet of terminal symbols,
- N is the ranked alphabet of nonterminal symbols,
- P is the set of production rules, described below, and
- $S \in N$ is the starting nonterminal

A production rule has the form $A \to f$, where A is a nonterminal and f an N-instantiated concatenation schema over Σ. We require that $rk(A) = otp(f)$ and that if f is a clone, then both its edges are labelled A.

A derivation step $g \to_p h$ for a production $A \to f = p \in P$ consists of replacing an edge marked with A in g with f, producing h. We write $\to_G$ for a derivation step using any of the rules of P, and $\to_G^*$ for the reflexive and transitive closure. We write $\mathcal{L}(G)$, indicating the *language* of the grammar G for the set of terminal graphs g such that $S^\bullet \to_G^* g$. If $g \to_{p_1} h' \to_{p_2} h$ and $g \to_{p_2} h'' \to_{p_1} h$, the two derivation steps are *independent*. Two derivations $d_1 = S^\bullet \to_G^* g$ and $d_2 = S^\bullet \to_G^* g$ are *distinct* if they cannot be made equal by reordering of independent derivation steps. Note that our view of derivation is essentially a linearised version of context-free derivation trees, where rule applications in different subtrees are independent, and distinct derivations have derivation trees that are distinguishable. However, the presence of cloning rules makes matters more involved, and Section 4 explains how these are handled.

An OPDG is *bottom-up deterministic* if, for each rule $A \to f$, there is no rule $B \to g$ such that $g \approx f$ and $B \neq A$. Informally, there are no two nonterminals that lead to the same right-hand side.

We conclude this section by sketching a parsing algorithm for OPDGs; for a detailed presentation, formal proofs, and complexity results, see [5]. In short, we can, without looking at the grammar, determine a number of useful properties of the input graph – in particular that there is an appropriate ordering of the leaves – and identify the graphs $g{\downarrow_x}$ for all nodes and edges x. Afterwards, assuming that the grammar is on normal form, we parse the graph bottom-up, marking each non-leaf node or edge x with the the nonterminals that could produce $g{\downarrow_x}$, and checking at each step which right-hand sides match. Finally, we check that the initial nonterminal is in the set of nonterminals that marks the root node.

3.3. Well-ordered DAG series

A commutative semiring is a tuple $\mathcal{C} = (C, +, \cdot, 0, 1)$ such that both $(C, \cdot, 1)$ and $(C, +, 0)$ are commutative monoids, $\cdot$ distributes over $+$, and $0 \cdot c = c \cdot 0 = 0$ for all $c \in C$. If, for every semiring element $c \in C$ except 0, there exists an element $c^{-1} \in C$ such that $c \cdot c^{-1} = 1$, then $\mathcal{C}$ is a commutative semifield. If $\mathcal{C}$ is a semifield and there also exists, for every $c \in C$, an element $-c \in C$ such that $c + (-c) = 0$, then $\mathcal{C}$ is a commutative field. The semiring is *zero-sum free* if there does not exists elements $a, b \in C \setminus \{0\}$ such that $a + b = 0$. It is *zero-divisor free* if there does not exists elements $a, b \in C \setminus \{0\}$ such that $a \cdot b = 0$.

By equipping OPDG rules with weights from a semiring, we can model weighted well-ordered DAG languages, in other words, well-ordered DAG series (WODS). A weighted OPDG (WOPDG) over commutative semiring $\mathcal{C}$ is a structure $G = (\Sigma, N, P, S, w)$, where (Σ, N, P, S) is an OPDG, and $w : P \to \mathcal{C}$ is the weight function.

The A-weight of a derivation $A^\bullet \to_{p_0} g_0 \to_{p_1} \ldots \to_{p_l} g$ is

$$\prod_i w(p_i) \ ,$$

and the weight of a graph is the sum of the weights of all distinct S-derivations that generate it. We generally call S-derivations derivations. The weight distribution thus defined is the

WODS $\mathcal{S}(G) : \mathcal{A}_\Sigma \to \mathcal{C}$. This means that if there is no (S-)derivation of g in G, then $\mathcal{S}(G)(g) = 0$. The *support* of a WOPDG G is the set of graphs $support(G) = \{g \mid \mathcal{S}(G)(g) \neq 0\}$. Note that the support of a WOPDG is a subset of the language of the underlying OPDG. If no rule is assigned weight 0, and the semiring is zero-sum and zero-divisor free, then the support of the WOPDG and the language of the underlying OPDG coincide. A WOPDG is *deterministic* if its underlying OPDG is. A WOPDG is *bottom-up deterministic* if, for every nonterminal A, there is at most one production rule $A \to g$ that has non-zero weight.

4. Initial algebra semantics

In this section, we establish a link between well-ordered DAG series and tree series, from which several results relating to minimisation (Section 5) and learnability (Section 6) immediately follow.

Definition 4.1 (Terms over concat_Σ)**.** *We associate with the set of concatenation operators* concat_Σ *the typed ranked alphabet* $\mathrm{concat}'_\Sigma = \{\hat{f} \mid f \in \mathrm{concat}_\Sigma\}$*, where* $rk(\hat{f})$ *equals the arity of* f *and* $tp(\hat{f})$ *is* $tp(f)$*.*

The terms over concat_Σ *is the set of trees* $\mathcal{T}_{\mathrm{concat}_\Sigma} \subset \mathbb{T}_{\mathrm{concat}'_\Sigma}$ *that are* type-matched *in the sense that for each subterm* $\hat{f}[t_1, \ldots, t_l]$*, the* i*th element of the argument type of* $\hat{f}$ *must match the output type of the root symbol of* t_i*.*

Let X *be the (infinite) typed ranked alphabet* $\{x_k \mid k \in \mathbb{N}\}$*, such that* $rk(x_k) = 0$ *and* $tp(x_k) = (k, \varepsilon)$ *for every* $k \in \mathbb{N}$*. Analogously to the tree case, the set* $\mathcal{T}_{\mathrm{concat}_\Sigma}(X)$ *is the set of type-matched trees over* $\mathrm{concat}'_\Sigma \cup X$

Terms over concat_Σ can be evaluated to yield graphs in $\mathcal{A}_\Sigma$. The construction is as expected. Evaluating a symbol $x_k \in X$ yields a placeholder edge with k targets.

Definition 4.2 (Term evaluation)**.** *The evaluation function* $eval : \mathcal{T}_{\mathrm{concat}_\Sigma}(X) \to \mathcal{A}_\Sigma$ *is defined as follows: For every* $x_k \in X$*,* $\mathrm{eval}(x_k)$ *is a single placeholder edge with exactly* k *targets, all external. For every* $t = \hat{f}[t_1, \ldots, t_k] \in \mathcal{T}_{\mathrm{concat}_\Sigma}(X) \setminus X$*,* $\mathrm{eval}(t) = f(\mathrm{eval}(t_1), \ldots, \mathrm{eval}(t_k))$*.*

The clones in concat_Σ need some special care, since their arguments have no inherent order. In what follows, we will write Cl to denote the set $\{\hat{f} \mid f \in \mathrm{concat}_\Sigma \wedge f \text{ is a clone}\}$ of all clones in concat_Σ.

Definition 4.3 (Top clone positions)**.** *Let* $t \in \mathcal{T}_{\mathrm{concat}_\Sigma}$*. The* top clone positions *of* t *is the set of positions*

$$\mathrm{cln}(t) = \{v \in \mathrm{pos}(t) \mid \text{ there is a path from } \mathrm{root}(t) \text{ to } v \text{ labelled } (Cl)^+(\mathrm{concat}'_\Sigma \setminus \{Cl\})\} \ .$$

The set of subtrees that attaches to the top clone positions in a term t can be freely permuted according to some bijection onto these positions, without affecting the value of t with respect to eval. This invariance induces an equivalence relation on $\mathcal{T}_{\mathrm{concat}_\Sigma}$.

Definition 4.4 (The relation $\sim$)**.** *The binary relation* $\sim$ *on* $\mathcal{T}_{\mathrm{concat}_\Sigma}$ *is defined as follows, for every* $t = \hat{f}[t_1, \ldots, t_k], s = \hat{g}[s_1, \ldots, s_n] \in \mathcal{T}_{\mathrm{concat}_\Sigma}$*:*

$$t \sim s \iff \hat{f} = \hat{g} \text{ and } \begin{cases} \exists \varphi \in \mathrm{biject}(\mathrm{cln}(t), \mathrm{cln}(s)) : \forall v \in \mathrm{cln}(t) : t/v \sim s/\varphi(v) & \text{if } \hat{f} \in Cl \\ t_i \sim s_i, \forall i \in [k] & \text{otherwise.} \end{cases}$$

It is straight-forward to show that $\sim$ *is an equivalence relation on* $\mathcal{T}_{\mathrm{concat}_\Sigma}$*.*

Lemma 4.5. *For every $g \in \mathcal{A}_\Sigma$, there is a tree $t \in \mathcal{T}_{\mathrm{concat}_\Sigma}$ such that $g = \mathrm{eval}(t)$, and t is unique modulo $\sim$.*

Proof. The proof is by induction on the size of $g{\downarrow_x}$, where $x \in V \cup E$. For the base case, assume that x is an edge and $g{\downarrow_x}$ has height one and thus consists of a single edge. There must then be a constant operation $f \in \mathrm{concat}_\Sigma$, such that $g{\downarrow_x} = f = \mathrm{eval}(\hat{f})$.

For the inductive case, first assume that $x \in V$. If x only has a single outgoing edge e, then $g{\downarrow_x} = g{\downarrow_e}$ and, since the inductive case for edges is handled below, we are done.

Assume that x has outgoing edges $\{e_1, \ldots, e_\ell\}$. Then $g{\downarrow_x}$ must be the result of a clone operator f applied to two smaller graphs h and h', which by the induction hypothesis can be uniquely represented (modulo $\sim$) as concatenation terms t and t', respectively. It follows that $g{\downarrow_x}$ can uniquely represented as $\hat{f}[t, t']$, as $\hat{f}[t, t'] \sim \hat{f}[t', t]$.

Next, assume that $x \in E$. Let $\mathrm{tar}(x) = v_1 \cdots v_k$ and let $v_{i_1} \cdots v_{i_\ell}$ be the non-leaf subsequence of $\mathrm{tar}(x)$. For each $j \in [\ell]$, by inductive assumption, the subgraph $g{\downarrow_{v_{i_j}}}$ is represented by a term t_{i_j} such that $\mathrm{eval}(t_{i_j}) = g{\downarrow_{v_{i_j}}}$, so $g{\downarrow_x}$ is represented by a term $\hat{f}[t_{i_j}, \ldots, t_{i_l}]$, for some suitable basic concatenation operator $f \in \mathrm{concat}_\Sigma$. □

Every ranked alphabet suggests a corresponding set of top concatenation operators.

Definition 4.6 (Top concatenation)**.** *Let Γ be a ranked alphabet. We denote by TOP_Γ the Γ-indexed family of* top-concatenations *$(c_\gamma)_{\gamma \in \Gamma}$, where for every $\gamma \in \Gamma$, c_γ is the top-concatenation with respect to γ.*

We extend the notion of top-concatenation to the domain $\mathcal{T}_{\mathrm{concat}_\Sigma}/\!\sim$ by letting

$$c_{\hat{f}}([t_1]_\sim, \ldots, [t_{rk(f)}]_\sim) \mapsto [\hat{f}[t_1, \ldots, t_{rk(f)}]]_\sim \ ,$$

for every $\hat{f} \in \mathrm{concat}'_\Sigma$ and $t_1, \ldots, t_{rk(f)} \in \mathcal{T}_{\mathrm{concat}_\Sigma}$. The function is well-defined, because for every $\hat{f} \in \mathrm{concat}'_\Sigma$, top concatenation with respect to $\hat{f}$ is a congruence with respect to $\sim$.

From Lemma 4.5 it follows that $\mathrm{eval} : \mathcal{A}_\Sigma \to \mathcal{T}_{\mathrm{concat}_\Sigma}/\!\sim$ is a bijection, and this gives us Theorem 4.7.

Theorem 4.7. *The algebras $(\mathrm{concat}_\Sigma, \mathcal{A}_\Sigma)$ and $(\mathsf{TOP}_{\mathrm{concat}'_\Sigma}, \mathcal{T}_{\mathrm{concat}_\Sigma}/\!\sim)$ are isomorphic.*

Theorem 4.7 suggests an alternative definition of ODPG semantics.

Definition 4.8. *Every WOPDG G over the alphabet Σ is a weighted tree grammar (wtg) over the typed ranked alphabet concat'_Σ. We denote by $\mathcal{S}_t(G)$ the tree series generated by G when viewed as a wtg.*

Definition 4.9 (Inital algebra semantics)**.** *Let $G = (\Sigma, N, P, S, w)$ be a WOPDG. The* initial algebra semantics *of G is the tree series $\mathcal{S}'(G) = \{(\mathrm{eval}(t), \mathcal{S}_t(G)(t)) \mid t \in \mathcal{S}_t(G)\}$.*

Observation 4.10. *For every WOPDG G, $\mathcal{S}(G) = \mathcal{S}'(G)$.*

A WOPDG is thus essentially a weighted tree grammar together with an evaluation function. This connection to tree series allows us to transfer a host of results.

5. Minimisation

In this section, we consider the minimisation problem for deterministic WOPDGs. We start by showing that if a grammar is bottom-up deterministic, then each graph in its support has a unique derivation tree. This is immediately implied by the following lemma.

Lemma 5.1. *Let $G = (\Sigma, N, P, S, w)$ be a WOPDG. Then G is bottom-up deterministic if and only if the following property holds. For every graph $g = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ in $\mathit{support}(G)$ and every $x \in V \cup E$ that is not a leaf node, there is a unique nonterminal $A \in N$ such that $A^\bullet \to_G^* g{\downarrow_x}$.*

Proof. The 'if' direction is immediate: if there were two distinct nonterminals that appeared as left-hand sides of rules with isomorphic right-hand sides, then there would be some graph that could be derived from both of them.

The 'only if' direction is proved by induction on, primarily, the height of $g{\downarrow_x}$, and secondarily, the outdegree of the root of $g{\downarrow_x}$. Since x is not a leaf, the base case is that x is an edge and the height of $g{\downarrow_x}$ is 1. Thus $g{\downarrow_x}$ is a single edge, together with its incident nodes. This means that for G to generate g, the subgraph $g{\downarrow_x}$ must be generated by a rule $A \to f$, where f is isomorphic to $g{\downarrow_x}$, for some $A \in N$. Since there cannot be two distinct nonterminals that generate graphs isomorphic to $g{\downarrow_x}$ and our grammars have no unit rules, A is the unique nonterminal such that $A^\bullet \to_G^* g{\downarrow_x}$.

For the inductive case, first assume that $x \in V$. If x has only a single outgoing edge e, then $g{\downarrow_x} = g{\downarrow_e}$ and, as the inductive case for edges are handled in the next paragraph, we are done. If, on the other hand, x has several outgoing edges $e_1, \ldots, e_\ell$, then we reason as follows. The only way for a node in a graph generated by an OPDG to have outdegree larger than one is if at some point in the derivation process, x was the source of a single nonterminal edge that was subsequently cloned. This means that there must be some nonterminal A such that each graph $g{\downarrow_{e_1}}, \ldots, g{\downarrow_{e_\ell}}$ can be generated by A and there is a clone rule in P for A. Furthermore, by inductive assumption, A is the unique nonterminal with this property. Thus A is also the unique nonterminal from which $g{\downarrow_x}$ can be derived.

Assume, finally, that x is an edge. Let $v_1, \ldots, v_\ell$ be the non-leaf targets of x. By inductive assumption, for each $i \in [\ell]$, there is a unique nonterminal A_i that can generate $g{\downarrow_{v_i}}$. Then $g{\downarrow_x}$ must have been generated starting with the application of a rule $A \to f$, where f is isomorphic to the graph obtained from $g{\downarrow_x}$ by replacing each graph $g{\downarrow_{v_i}}$ by a single edge labeled A_i, attached to the sequence of leaf nodes of $g{\downarrow_x}$ that are external for $g{\downarrow_{v_i}}$. Since no distinct nonterminals can appear in rules with isomorphic right-hand sides, A must be the unique such nonterminal. □

Another way of stating Lemma 5.1 is the following. For each $A \in N$, let G_A be the grammar obtained from G by replacing S with A as starting symbol. Then G is bottom-up deterministic if and only if, for each pair A_1 and A_2 of nonterminals from N, $\mathit{support}(G_{A_1}) \cap \mathit{support}(G_{A_2}) \neq \emptyset$ implies $A_1 = A_2$. In other words, the concept of bottom-up determinism coincides with the notion of unambiguity for OPDGs, as defined in [4]. Thus we can restate one of the results from that article:

Theorem 5.2 (cnf. [4])**.** *If $\mathcal{S}$ is a series generated by some deterministic WOPDG over a commutative semifield, then there is a unique (up to isomorphism) minimal deterministic WOPDG G_L such that $\mathcal{S}(G_L) = \mathcal{S}$.*

Theorem 5.2 ensures that the *minimisation problem* for deterministic WOPDGs always has a unique solution, modulo nonterminal names. The problem is stated as follows:

Definition 5.3 (Minimization problem)**.** *Given a deterministic WOPDG G, find the unique minimal deterministic WOPDG for $\mathcal{S}(G)$.*

Rather than formulating a minimisation algorithm that solves Problem 5.3 directly, we show that the problem can be reduced to finding the unique minimal weighted deterministic regular tree grammar for $\mathcal{S}_t(G)$. For this purpose, we note that the forward or backward application of eval does not affect the nonterminal to which a tree or DAG is mapped:

Lemma 5.4. *Let G be a deterministic WOPDG. For every non-terminal A in G and every $t \in \mathcal{T}_{\mathrm{concat}'_\Sigma}$, $\mathcal{S}_t(G_A)(t) = \mathcal{S}(G_A)(\mathrm{eval}(t))$.*

In preparation for the proof of Lemma 5.6, we lift the notion of *contexts* from the tree domain to the graph domain. Intuitively, a context is a graph with a single, appropriately placed placeholder edge.

Definition 5.5 (Graph context)**.** *A* graph context *over Σ is the evaluation of a tree in $\mathcal{T}_{\mathrm{concat}_\Sigma}(X)$ with a single occurrence of a symbol x_k from X.*

This yields a graph context c of some type (m, k) with a single placeholder edge e of rank k. We can substitute a graph $f \in \mathcal{A}_\Sigma$ of appropriate type (k, ε) into c in the standard way, yielding the graph $g = c[\![e : f]\!]$ of type (m, ε). We also write this operation as $c[\![f]\!]$.

It is straightforward to show that taking a graph $g \in \mathcal{A}_\Sigma$ and replacing $g{\downarrow_x}$ for some $x \in V_g \cup E_g$ with a placeholder edge of rank $|g{\downarrow_x}_{\bullet\bullet}|$ yields a graph context in this sense.

Lemma 5.6. *Let G be a deterministic WOPDG over a commutative semifield, and H the minimal deterministic weighted tree grammar for $\mathcal{S}_t(G)$. Then H is the minimal deterministic WOPDG for $\mathcal{S}(G)$.*

Sketch. The nonterminals A and B in G are *distinguishable w.r.t.* $\mathcal{S}(G)$ if there is a graph context c and graphs $g \in \mathit{support}(\mathcal{S}(G_A))$ and $h \in \mathit{support}(\mathcal{S}(G_B))$ such that

$$\mathcal{S}(G)(c[\![g]\!]) \cdot \mathcal{S}(G_A)(g)^{-1} \neq \mathcal{S}(G)(c[\![h]\!]) \cdot \mathcal{S}(G_B)(h)^{-1} \ .$$

Similarly, the pair of nonterminals is *distinguishable w.r.t.* $\mathcal{S}_t(G)$ if there is a tree context c and trees $t \in \mathit{support}(\mathcal{S}_t(G_A))$ and $s \in \mathit{support}(\mathcal{S}_t(G_B))$ such that

$$\mathcal{S}_t(G)(c[\![t]\!]) \cdot \mathcal{S}_t(G_A)(t)^{-1} \neq \mathcal{S}_t(G)(c[\![s]\!]) \cdot \mathcal{S}_t(G_B)(s)^{-1} \ .$$

We first ensure that H is a WOPDG. Since H is minimal for $\mathcal{S}_t(G)$, and both H and G are deterministic, the nonterminals of H can be obtained by merging every set of mutually indistinguishable nonterminals w.r.t. $\mathcal{S}_t(G)$ into a single nonterminal [16]. This means in particular that every clone rule in H can be written in the form $P \to f[P, P]$, where P is an equivalence class of mutually indistinguishable nonterminals. Moreover, to see that the merge respects $\sim$, we argue as follows: From Theorem 4.7 we have that $t \sim s$ implies $\mathrm{eval}(t) = \mathrm{eval}(s)$, and by Lemma 5.4 that there is a nonterminal A such that $t, s \in \mathit{support}(\mathcal{S}_t(G_A))$, so there is a nonterminal $B \in H$ (that is the result of merging A the with indistinguishable nonterminals) such that $t, s \in \mathit{support}(\mathcal{S}_t(H_B))$. It follows that H is a valid WOPDG, and that $\sim$ is a congruence with respect to H.

It is then straight-forward to show that for every WOPDG G and nonterminals A and B in G, A and B are distinguishable w.r.t. $\mathcal{S}(G)$ if and only if they are distinguishable w.r.t. $\mathcal{S}_t(G)$, so H is minimal also for $\mathcal{S}(G)$. A witness context for the distinguishability of a pair of nonterminals in one domain, can be translated to a witness context in the other, by extending eval and eval^{-1} to tree and graph contexts in the expected way. □

Lemma 5.6 means that the minimisation result established in [7] and [16] are directly transferable to our setting. In the statement of these results, we assume a deterministic input WOPDG G over different types of semirings and let r denote the maximal number of non-terminals in the right-hand side of any production of G, m denote the size of the input grammar G, and n denote the number of nonterminals in G. A WOPDG is *all-accepting* if assigns a non-zero value to every graph in its domain.

Theorem 5.7 (cnf. Theorem 4.12 of [7])**.** *The minimisation problem for all-accepting deterministic WOPDGs over commutative fields is solvable in $O(rm \log n)$.*

Theorem 5.8 (cnf. [16])**.** *The minimisation problem for deterministic WOPDGs over commutative semifields is solvable in $O(rmn)$.*

6. MAT Learning

The relation between WODS and tree series established in Section 4 also makes results on grammatical inference for tree languages transferable to our DAG domain. Here, we focus on the Minimal Adequate Teacher (MAT) model due to Angluin [1]. The MAT model supposes two entities – a *learner* and a *teacher*. The teacher already knows the target series $\mathcal{S}$, and it is the objective of the learner to infer $\mathcal{S}$. The learner gathers information about $\mathcal{S}$ by querying the teacher: In a *coefficient query*, the learner gives the teacher a graph g, and the teacher answers with the weight $\mathcal{S}(g)$. In an *equivalence query*, the learner gives the teacher a WOPDG G. If G represents $\mathcal{S}$ correctly, then the teacher confirms the successful inference and the learning ends. If not, the teacher returns a *counterexample* – a graph g that is assigned an erroneous weight by G, i.e., that is such that $\mathcal{S}(G)(g)$ and $\mathcal{S}(g)$ differ.

Definition 6.1 (MAT learning)**.** *A MAT teacher for a WODS $\mathcal{S}$ over a semiring $\mathcal{C}$ is an oracle capable of answering two types of queries:*

- ***Coefficient queries**: Given $g \in \mathcal{A}_\Sigma$, what is $\mathcal{S}(g)$?*
- ***Equivalence queries**: Given a WOPDG G, is $\mathcal{S}(G) = \mathcal{S}$? If yes, the teacher confirms the successful inference of $\mathcal{S}$, if no, the teacher returns a counterexample, that is, a graph $g \in \mathcal{A}_\Sigma$ such that $\mathcal{S}(G)(g) \neq \mathcal{S}(g)$.*

A class of graph series is *MAT learnable* if every series in the class can be inferred within the MAT model. In general, this is true for classes for which there is a Myhill-Nerode theorem, such as recognisable string and tree series [9]. As we shall see, WODLs and WODSs over commutative semifields also meet this description. Rather than providing an explicit MAT-learning algorithm for the latter class, we show that the problem of inferring a target WODS $\mathcal{S}$ over the commutative semifield $\mathcal{C}$ can be reduced to that of inferring a regular tree series $\mathcal{S}_t$ over $\mathcal{C}$, and that $\mathcal{S}_t$ is easily derivable from $\mathcal{S}$.

Definition 6.2 (The series $\mathcal{S}_t$)**.** *Let $\mathcal{S} : \mathcal{A}_\Sigma \to \mathcal{C}$ be a WODS. The regular tree series $\mathcal{S}_t : \mathbb{T}_{\mathrm{concat}'_\Sigma} \to \mathcal{C}$ is given by*

$$\mathcal{S}_t(t) = \begin{cases} \mathcal{S}(g) & \text{if } t \in \mathit{eval}^{-1}(g) \text{ for some } g \in \mathit{support}(\mathcal{S}) \text{ , and} \\ 0 & \text{otherwise.} \end{cases}$$

In preparation for Theorem 6.3, we recall the MAT learner for tree series over commutative semifields given in [15] (there formulated for weighted tree automata, see also [13]). In the inference of a series $\mathcal{S}$, the learner gathers two sets of trees, S and T. Both are subtree-closed in the sense that if they contain a tree t, then they also contain every subtree of t. Additionally, every direct subtree of a tree in T is contained in S. The purpose of S is to collect representatives of the syntactic congruence classes with respect to $\mathcal{S}$, which are in a one-to-one correspondence with the nonterminals of the minimal wtg G_t that generates $\mathcal{S}_t$. To avoid confusion, we write $\langle t \rangle$ to express that a tree t is viewed as a nonterminal.

The learner also maintains an auxiliary set of contexts E that witness (i) that every tree in S is a subtree of some tree in $\mathit{support}(\mathcal{S}_t)$, and (ii) that the trees in S are syntactically distinct. The purpose of T is to represent production rules of the hypothesis grammar, and a tree $f[t_1, \ldots, t_k] \in T$ encodes the production rule $\langle \mathit{rep}(t) \rangle \to f(\langle t_1 \rangle, \ldots, \langle t_k \rangle)$, where $\mathit{rep}(t)$ is the unique tree in S such that $\mathit{rep}(t)$ and $f[t_1, \ldots, t_k]$ are indistinguishable with respect to the contexts in E (if no such tree exists, then $f[t_1, \ldots, t_k]$ is added to S).

From S and T, the learner synthesises a weighted tree grammar H that is passed to the teacher through an equivalence query. The wtg has the property that every tree $t \in T$ is

in $\mathit{support}(\mathcal{S}_t)(H_{\langle \mathit{rep}(t)\rangle})$, where $H_{\langle \mathit{rep}(t)\rangle}$ is the grammar obtained from H by replacing the initial nonterminal by $\langle \mathit{rep}(t)\rangle$.

The learner collects the elements of S and T by processing the teacher's counterexamples through *contradiction-backtracking* [17]. This essentially consists in step-wise simulation of the parsing of a counterexample t with respect to the current hypothesis wtg H. The learner repeatedly selects a subtree of t on the form $f[t_1, \ldots, t_k]$ for some $t_1, \ldots, t_k \in S$. If this subtree is not in T, then it is added to T, and the learner has found a new production rule. If it is in T, it is replaced in t by $\mathit{rep}(f[t_1, \ldots, t_k])$. The learner then uses a coefficient query to verify that the counterexample is still a counterexample. If it is not, then the learner has discovered that $f[t_1, \ldots, t_k]$ and $\mathit{rep}(f[t_1, \ldots, t_k])$ belong to different syntactic congruence classes, i.e., the learner has found a new nonterminal. Since the learner disagrees with the teacher about t, it is guaranteed to find at least one new transition or nonterminal by backtracking, and this guarantees that the overall inference process eventually terminates.

Theorem 6.3. *WODSs over commutative semifields are MAT learnable in polynomial time.*

Proof. We show that the problem of inferring a target WODS $\mathcal{S}$ can be reduced to that of inferring the tree series $\mathcal{S}_t$ defined above, and then applying the existing MAT-learning algorithm for trees series over semifields given in [15]. We henceforth refer to this algorithm as the learner. With this approach, the problem becomes one of finding a way to simulate a MAT teacher for $\mathcal{S}_t$ using the available MAT teacher for $\mathcal{S}$.

For coefficient questions, the simulation is easy. When the learner wants to ask a coefficient question for the tree $t \in \mathbb{T}_{\mathrm{concat}'_\Sigma}$, we first check if it is type matched. If not, we answer the learner that it has weight 0. If it is type matched, then $\mathrm{eval}(t)$ is well-defined, so we ask the teacher a coefficient question for $\mathrm{eval}(t)$, and as the answer holds equally for every $s \in \mathrm{eval}^{-1}(\mathrm{eval}(t))$, it holds in particular for t.

To simulate equivalence queries, we must ensure that the hypothesis grammar H maintained by the learner is not only a well-formed weighted tree grammar, but also a well-formed WOPDG. This requires us to argue that the following invariants hold:

- First, we require the trees produced be type-matched, and that for each rule $A \to f$, that $\mathrm{rank}(A) = \mathit{otp}(\mathit{rep}(f)) = \mathit{otp}(f)$
- Second, all clone rules in H are on the form $A \to f[A, A]$ for some nonterminal A and clone f of appropriate type.

We deal with these in order:

- Note that for every tree $s \in S$, the learning algorithm in [15] is guaranteed to have collected at least one context $c \in E$, such that $\mathrm{eval}(c[\![s]\!]) \in \mathit{support}(\mathcal{S})$, meaning s is type-matched. Moreover, each tree $t \in T$ can be freely substituted for its representative $\mathit{rep}(t) \in S$ in all of the contexts $c \in C$, so by the same reasoning, all trees in T are type-matched. That is, for each tree $f[s_1, \ldots, s_\ell] \in T$ with $s_i \in S$ for all $i \in [\ell]$, $\mathit{otp}(s_i) = \mathit{atp}(f)_i$. Finally, by the substitution of t for s in c, $\mathit{otp}(t) = \mathit{otp}(s)$.
- As described in [15], the learner is reactive in its collection of trees that represent productions, and will therefore only include clone rules in H if it has come across such in the contradiction-backtracking on some tree t on the form $c[\![f[s, s']]\!]$ that is in $\mathit{support}(\mathcal{S})$ but not in $\mathit{support}(\mathcal{S}(H))$. When this happens, the learner uses contradiction backtracking and incrementally replaces larger and larger subtrees of s and s' by trees that is in its set of nonterminal representatives S, but if it any points produces a tree $c'[\![f[r, r']]\!]$ such that exactly one of r or r' is not in the same syntactic

congruence class as s and s', then $c'[\![f[r,r']]\!]$ will fall outside of $support(\mathcal{S})$, and the algorithm will learn that r and r' are not syntactically equivalent.

Hence, by the time the learner has replaced s and s' with trees in S, it has to be the same tree, because it has only one representative per syntactic class, so the example will be on the form $c''[\![f[r,r]]\!]$ for some $r \in S$. Since $f[r,r]$ will then be in the set of representatives for production rules T, since it is in the same syntactic class as r, and since the grammar H produced by the learner is guaranteed to map every tree in T to the correct nonterminal in S, $f[r,r]$ will be taken to r, and thus represents a valid clone rule.

We can thus pass H to the teacher through equivalence queries, and if we are given in return a counterexample g, then any tree in $\text{eval}^{-1}(g)$ is a counterexample for the learner, because H maps each tree in $\text{eval}^{-1}(g)$ to the same nonterminal. □

7. Logical characterisation

Our aim in this section is to prove that OPDL is a set of MSO definable graph languages. Thus, given an OPDG $G = (\Sigma, N, P, S)$, our goal is to construct an MSO formula φ_G such that for every graph g,

$$g \in \mathcal{L}(G) \Leftrightarrow g \models \varphi_G.$$

Let r be the maximal rank of any edge appearing in G. The vocabulary we use for φ_G has the following predicate symbols:

Node $\text{Node}(v)$ is true if v is a node.

Edge $\text{Edge}(e)$ is true if e is an edge.

Src $\text{Src}(e, v)$ is true if e is an edge and v is the source node of e.

Tar^i For every $i \in [r]$, $\text{Tar}^i(e, i)$ is true if e is an edge and v is the ith target node of e.

Lab$_a$ For every $a \in \Sigma \cup N$, $\text{lab}_a(e)$ is true if e is an edge and a is the label of e.

Ext $\text{Ext}(v)$ is true if v is a node and v is external.

We can now define the following useful formulas:

$$\begin{aligned}
\text{Tar}(e,v) &= \bigvee_{i\in[r]} \text{Tar}^i(e,v) \\
\text{Root}(v) &= \text{Node}(v) \wedge \neg\exists e(\text{Tar}(e,v)) \\
\text{Leaf}(v) &= \text{Node}(v) \wedge \neg\exists e(\text{Src}(e,v)) \\
\text{Indegree}(v) = 1 &= \exists e(\text{Tar}(e,v) \wedge (\forall x(\text{Tar}(x,v) \rightarrow x = e))) \\
\text{Outdegree}(v) = 1 &= \exists e(\text{Src}(e,v) \wedge (\forall x(\text{Src}(x,v) \rightarrow x = e)))
\end{aligned}$$

We leave the somewhat tedious construction of a formula that ensures that the input structure correctly encodes a well-ordered DAG for the Appendix, where the formula WDAG is defined.

For every rule $\rho = A \rightarrow f \in P$ that is not a clone rule, we want to define a formula $\text{P}_\rho(e, x_0, x_1, \ldots, x_k)$ that is true for a subgraph $g{\downarrow_e}$ if the following conditions are met:

1. x_0 is the unique root of $g{\downarrow_e}$ and e the unique edge in $g{\downarrow_e}$ connected to x_0.

2. The external nodes of $g{\downarrow_e}$ are $x_0 x_1 \cdots x_k$

3. $g{\downarrow_e}$ can be derived from A in G, starting with an application of ρ.

We next describe how to construct such formulas. In the construction, we use the formula $\mathrm{Reent}^i(x, v)$, that is true if and only if v is the ith reentrant node of edge x (the formula is defined in the Appendix). Further, let P_A be the subset of P having A as its left-hand side, for all $A \in N$.

Given $\rho = A \to f$, we assume without loss of generality that

- v_0 is the unique root of f and f_0 is the unique edge connected to v_0.
- $\mathrm{lab}_f(f_0) = a$
- $f_1, \ldots, f_\ell$ is the ordered sequence of nonterminal edges in f.

We arbitrarily number the non-root nodes in f and call them $v_1, \ldots, v_n$. We start the formula $\mathrm{P}_\rho(e, x_0, x_1, \ldots, x_k)$ by existentially quantifying over $n+1$ variables $y_0, y_1, \ldots, y_n$, corresponding to $v_0, v_1, \ldots, v_n$. Then we simply describe the structure of f with a conjunction of the following kinds of formulas:

- $\mathrm{Src}(e, y_0) \wedge \mathrm{Lab}_f(e)$
- If v_i is the jth target of f_0, then $\mathrm{Tar}^j(e, y_i)$.
- If v_i is the jth node in ext_f, then $y_i = x_j$.
- If v_i is the source of $f' \neq f_0$, and v_t is the jth target of f', then $\mathrm{Reent}^j(v_i, v_t)$.

The last item is motivated by the fact that in g, the nonterminal edges have been replaced by graphs, whose reentrancies are exactly the targets of the nonterminal edges.

We also need to add the recursive requirement that the graphs that have replaced the nonterminals of f could have been derived from them. To this end, let v_i be the source of f_j. We then need to state that for any edge d that has v_i as its source in g, the subgraph $g{\downarrow_d}$ could have been produced from f_j, or, more succinctly, that $g{\downarrow_d} \in \mathcal{L}(G_A)$ for $A = \mathrm{lab}(f_j)$.

Let f_j have rank s and let $\mathrm{tar}_f(f_j) = v_{i_1} \cdots v_{i_s}$. We then require that for any such edge d, the formula $P_{\rho'}(d, y_i, y_{i_1}, \ldots, y_{i_s})$ should hold, for some rule $\rho' \in P_A$. Furthermore, if $\mathrm{lab}_f(f_j)$ is not clonable, we require that y_i has outdegree 1 in g.

Note that the recursion involved in the rules will, for graphs generated by G, always terminate with rules where the right-hand side has no nonterminals.

Now that we have the formulas P_ρ, for all $\rho \in P$, we are ready to define the formula L_G that states that the input structure is a graph in the language of G. Let i be the rank of the start nonterminal S. We want to state that the input is a DAG and that it can be produced from S. In order to do so, we need to identify the root. Depending on whether or not S is clonable, we get a different variant of the formula. In the non-clonable case, we get

$$L_G = \mathrm{WDAG} \wedge \forall v(\mathrm{Root}(v) \to (\mathrm{Outdegree}(v) = 1 \wedge \exists e, u_1, \ldots, u_i(\mathrm{Src}(e, v) \wedge \bigvee_{\rho \in P_S} \mathrm{P}_\rho(e, v, u_1, \ldots, u_i)))).$$

In the case where S is clonable, the formula instead looks as follows.

$$L_G = \mathrm{WDAG} \wedge \forall v(\mathrm{Root}(v) \to (\exists u_1, \ldots, u_i(\forall e(\mathrm{src}(e, v) \to \bigvee_{\rho \in P_S} \mathrm{P}_\rho(e, v, u_1, \ldots, u_i))))).$$

The formula L_G is closed, and will be true for exactly those input structures that represent DAGs generated by G. We note that we have used no second-order quantification in the above, but it is needed in the definitions of WDAG and the formulas Reent^i, stated in the Appendix.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[2] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. Abstract meaning representation for sembanking. In *7th Linguistic Annotation Workshop & Interoperability with Discourse, Sofia, Bulgaria*, 2013.

[3] Martin Berglund, Henrik Björklund, and Frank Drewes. Single-rooted DAGs in regular DAG languages: Parikh image and path languages. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+13)*, pages 94–101, 2017.

[4] H. Björklund, B. Björklund, and P. Ericson. On the regularity and learnability of ordered DAG languages. In Arnaus Carayol and Cyril Nicaud, editors, *22nd International Conference on the Implementation and Application of Automata (CIAA 2017), Marne-la-Vallée, France*, volume 10329 of *Lecture Notes in Computer Science*, pages 27–39. Springer International Publishing, 2017.

[5] H. Björklund, F. Drewes, and P. Ericson. Between a rock and a hard place – uniform parsing for hyperedge replacement DAG grammars. In *10th International Conference on Language and Automata Theory and Applications (LATA 2016), Prague, Czech Republic, 2016*, pages 521–532, 2016.

[6] Henrik Björklund, Frank Drewes, Petter Ericson, and Florian Starke. Uniform parsing for hyperedge replacement grammars. Technical Report UMINF 18.13, Umeå University, http://www8.cs.umu.se/research/uminf/index.cgi, 2018. Submitted for publication.

[7] Johanna Björklund, Andreas Maletti, and Jonathan May. Bisimulation minimisation for weighted tree automata. In *Proceedings of the 11th International Conference on Developments in Language Theory (DLT 2007), Turku, Finnland*, volume 4588 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2007. Springer Verlag.

[8] Johannes Blum and Frank Drewes. Properties of regular DAG languages. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *LATA*, volume 9618 of *Lecture Notes in Computer Science*, pages 427–438. Springer, 2016.

[9] Björn Borchardt. The Myhill-Nerode theorem for recognizable tree series. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, pages 146–158, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[10] D. Chiang, J. Andreas, D. Bauer, K. M. Hermann, B. Jones, and K. Knight. Parsing graphs with hyperedge replacement grammars. In *51st Annual Meeting of the Association for Computational Linguistics (ACL 2013), Sofia, Bulgaria*, pages 924–932, 2013.

[11] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. Weighted DAG automata for semantic graphs. *Computational Linguistics*, 44(1):119–186, 2018.

[12] F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars*, volume 1, pages 95–162. World Scientific, 1997.

[13] Frank Drewes and Heiko Vogler. Learning deterministically recognizable tree series. *Journal of Automata, Languages and Combinatorics*, 12(3):332–354, 2007.

[14] S. Gilroy, A. Lopez, S. Maneth, and P. Simonaitis. (Re)introducing regular graph languages. In *Proceedings of the 15th Meeting on the Mathematics of Language (MOL 2017)*, pages 100–113, 2017.

[15] Andreas Maletti. Learning deterministically recognizable tree series — revisited. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics*, pages 218–235, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[16] Andreas Maletti. Minimizing deterministic weighted tree automata. *Information and Computation*, 207(11):1284–1299, 2009.

[17] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

Appendix

In this section, we define the formulas $\mathrm{Reent}^i(e, v)$ and $\mathrm{Reent}(e, X)$, as well as the formula WDAG, stating that the input structure is a well-ordered DAG.

Basic graph properties

We define formulas stating that there is a single root (and thus that the graph is connected and fully reachable from said root), that each element of the universe is either an edge or a node, and that each edge has a unique label, a unique source, and the correct number of targets.

$$
\begin{aligned}
\text{Single-Rooted} &= \exists r(\mathrm{Root}(r) \wedge \forall u(\mathrm{Root}(u) \rightarrow (u = r))) \\
\text{Partition} &= \forall x(\mathrm{Edge}(x) \leftrightarrow \neg\mathrm{Node}(x)) \\
\text{UniqueLabels} &= \forall x(\mathrm{Edge}(x) \rightarrow (\bigvee_{a\in\Sigma} (\mathrm{Lab}_a(x) \bigwedge_{b\in\Sigma\setminus\{a\}} \neg\mathrm{Lab}_b(x))) \wedge \\
&\quad (\neg\mathrm{Edge}(x) \rightarrow \bigwedge_{a\in\Sigma} \neg\mathrm{Lab}_a(x))) \\
\text{UniqueSources} &= \forall e(\mathrm{Edge}(e) \rightarrow \exists v(\mathrm{Src}(e, v) \wedge \forall u(\mathrm{Src}(e, u) \rightarrow (u = v)))) \wedge \\
\text{Targets} &= \forall e(\mathrm{Edge}(e) \rightarrow \bigvee_{a\in\Sigma} (\mathrm{Lab}_a(e) \wedge \\
&\quad \bigwedge_{i\in[\mathrm{rank}(a)]} (\exists v(\mathrm{Tar}^i(e, v) \wedge \forall u(\mathrm{Tar}^i(e, u) \rightarrow (u = v))))) \wedge \\
&\quad \bigwedge_{j\in[r]\setminus[\mathrm{rank}(a)]} (\neg\exists v(\mathrm{Tar}^j(e, v))))
\end{aligned}
$$

We also define a formula Externals that makes sure that the root is external, the rest of the external nodes are leafs, and, by enumeration, that they are as many as the rank of the start nonterminal. With this in hand, we define the formula Graph, that simply ensures all of the above properties:

$$\mathrm{Graph} = \text{Single-Rooted} \wedge \mathrm{Partition} \wedge \mathrm{UniqueLabels} \wedge \mathrm{UniqueSources} \wedge \mathrm{Targets} \wedge \mathrm{Externals}$$

Reachability and reentrancies

In order to be able to speak about reachability, we define the notion of a set being closed under the Src and Tar relations:

$$\mathrm{Closed}(S) = \forall x \forall y(x \in S \wedge (\mathrm{Src}(y, x) \vee \mathrm{Tar}(x, y)) \rightarrow y \in S)$$

In other words, if $x \in S$, then any edge or node reachable from x also belongs to S.

Towards the definitions of reentrancies and ordering we now define directed reachability.

$$\mathrm{ReachableSet}(x, Y) = \forall S((\mathrm{Closed}(S) \wedge x \in S) \rightarrow \forall y(y \in Y \rightarrow y \in S))$$

The above formula makes sure that Y is the smallest closed set that contains x, or, in other words, that Y is the set of nodes and edges reachable from x. For convenience, we also define reachability between individual nodes and edges:

$$\mathrm{Reachable}(x, y) = \exists Y(\mathrm{ReachableSet}(x, Y) \wedge y \in Y)$$

We are now ready to define the set of reentrant nodes with respect to a node or edge:

$$\begin{aligned}\text{Reent}(x, X) = \forall y(y \in X \leftrightarrow (\text{Reachable}(x, y) \wedge \text{Leaf}(y) \wedge \\ (\text{Ext}(y) \vee \exists z(\neg\text{Reachable}(x, z) \wedge \neg\text{Reachable}(z, x) \wedge \text{Reachable}(z, y)))))\end{aligned}$$

The formula says that for y to be reentrant with respect to x, it has to be a leaf reachable from x and either be external or also reachable from some z such that z is not reachable from x and x is not reachable from z. This is also a sufficient condition.

We require that all edges have the same set of reentrant nodes as their sources. This neatly covers the requirement that edges sharing the same source have the same set of reentrant nodes.

$$\begin{aligned}\text{ReentClones} = \forall e, v(\text{Src}(e, v) \rightarrow \\ \forall x, X, Y(\text{Reent}(e, X) \wedge \text{Reent}(v, Y) \rightarrow (x \in X \leftrightarrow x \in Y)))\end{aligned}$$

Ordering

We can define the notion of closest common ancestor edges, as follows:

$$\text{CommonAncestor}(x, u, v) = \text{Reachable}(x, u) \wedge \text{Reachable}(x, v)$$

$$\begin{aligned}\text{CCAE}(e, u, v) = \text{Edge}(e) \wedge \text{CommonAncestor}(e, u, v) \wedge \\ \neg\exists w(\text{Tar}(e, w) \wedge \text{CommonAncestor}(w, u, v))\end{aligned}$$

To define the ordering of leaves, we have the following.

$$\begin{aligned}\text{Before}(u, v) = \text{Leaf}(u) \wedge \text{Leaf}(v) \wedge ((u = v) \vee \forall e(\text{CCAE}(e, u, v) \\ \rightarrow \exists x \forall y(\text{Reachable}(x, u) \wedge \text{Reachable}(y, v) \wedge \text{Tar}^i(e, x) \wedge \text{Tar}^j(e, y) \rightarrow i < j)))\end{aligned}$$

This is a shorthand, as we cannot directly compare i and j. However, as the alphabet has bounded rank, the above can be achieved by a disjunction over all relevant pairs of values for i and j.

Now, to make sure we have a consistent ordering, we use the following formula.

$$\text{ConsistentOrdering} = \forall e, u, v((\text{CCAE}(e, u, v) \wedge u \neq v) \rightarrow (\text{Before}(u, v) \leftrightarrow \neg\text{Before}(v, u)))$$

Finally, we need to make sure that the graph is really a DAG:

$$\text{DAG} = \forall x, y(\text{Reachable}(x, y) \wedge \text{Reachable}(y, x) \rightarrow x = y)$$

In conclusion, our precondition amounts to

$$\text{WDAG} = \text{Graph} \wedge \text{ReentClones} \wedge \text{ConsistentOrdering} \wedge \text{DAG}$$

Given a consistent ordering, we can find the ordering of the reentrant nodes, and in particular if a specific leaf v is the ith member of the sequence of reentrant leaves of some edge or node x.

$$\text{Reent}(x, y) = \forall X(\text{Reent}(x, X) \rightarrow y \in X)$$

$$\begin{aligned}\text{Reent}^i(x, v) = \text{Reent}(x, v) \wedge \exists u_1, \ldots u_i(\bigwedge_j \text{Reent}(x, u_j) \wedge \bigwedge_j \text{Before}(u_j, v) \wedge \\ \bigwedge_{j \neq k}(u_j \neq u_k) \wedge \forall y(\text{Reent}(x, y) \wedge \text{Before}(y, x) \rightarrow \bigvee_j (y = u_j)))\end{aligned}$$

The last formula simply enumerates i nodes that are reentrant for x, the last of which is v. This concludes the definition of the formulas used in Section 7.

IV

Uniform Parsing for Hyperedge Replacement Grammars

Henrik Björklund[a], Frank Drewes[a], Petter Ericson[a], Florian Starke[b]

[a] *Deptartment of Computing Science, Umeå University, Sweden*
[b] *Faculty of Computer Science, TU Dresden, Germany*

Abstract

It is well known that hyperedge-replacement grammars can generate NP-complete graph languages even under seemingly harsh restrictions. This means that the parsing problem is difficult even in the non-uniform setting, in which the grammar is considered to be fixed rather than being part of the input. Little is known about restrictions under which truly uniform polynomial parsing is possible. In this paper we propose a low-degree polynomial-time algorithm that solves the uniform parsing problem for a restricted type of hyperedge-replacement grammars which we expect to be of interest for practical applications.

1. Introduction

Hyperedge-replacement grammars (HR grammars, for short) are context-free graph grammars that were introduced in (Bauderon and Courcelle, 1987; Habel and Kreowski, 1987), see also Habel (1992); Drewes et al. (1997). They represent one of the two most successful formal models for the description of graph languages (the other being confluent node-replacement grammars), because of their favorable algorithmic and language-theoretic properties which closely resemble those of context-free string grammars. Unfortunately, the similarities between the string and graph cases fail to extend to one of the most important computational problems in the context of formal languages: the parsing problem. It has been known for a long time that even the non-uniform membership problem for context-free graph languages is intractable (unless $\mathrm{P} \neq \mathrm{NP}$). In particular, there are hyperedge replacement graph languages which are NP-complete (Aalbersberg et al., 1986; Lange and Welzl, 1987). Severe restrictions must be placed on the grammars in order to make at least non-uniform polynomial parsing possible. Early results in this regard can be found in (Lautemann, 1990; Vogler, 1991; Drewes, 1993b). In (Lautemann, 1990) the degree of the polynomial that bounds the running time varies with the language. The algorithm in (Vogler, 1991), which considers only edge replacement, and its generalization to hyperedge replacement by Drewes (1993b) are cubic in the size of the input graph, but depend exponentially on the grammar if considered in a uniform setting. Moreover, the restrictions Vogler (1991) and Drewes (1993b) placed on the considered

Email addresses: henrikb@cs.umu.se (Henrik Björklund), drewes@cs.umu.se (Frank Drewes), pettter@cs.umu.se (Petter Ericson), Florian.Starke@tu-dresden.de (Florian Starke)

graph languages are very strong, and it was shown in Drewes (1993a) that even a slight relaxation results in NP-completeness again. For these reasons, these parsing algorithms are mainly of theoretical interest.

In recent years the question of efficiently parsing hyperedge replacement languages received renewed interest, because hyperedge replacement was proposed as a suitable mechanism for describing sentence semantics in natural language processing, and in particular the abstract meaning representation proposed in Banarescu et al. (2013). Regarding the use of hyperedge replacement in this application area, see Chiang et al. (2013). The same paper described a general recognition algorithm together with a detailed complexity analysis. Unsurprisingly, the running time of the algorithm is exponential even in the non-uniform case, one of the exponents being the maximum degree of nodes in the input graph. The same is true for the recent algorithm by Gilroy et al. (2017) which implements parsing for so-called *regular tree grammars*.

Unfortunately, the node degree is one of the parameters one would ideally not wish to limit, since meaning representations do not have bounded node degree. Moreover, natural language processing often has to deal with algorithmic learning situations in which large corpora must be parsed and grammars adjusted in an iterative process. Thus, truly uniform polynomial-time solutions would be valuable, provided that the polynomials have a reasonably low degree and the restrictions on the grammars are "natural".

Parsing a graph G with respect to a given HR grammar $\mathcal{G}$ means to check whether there is a derivation tree in $\mathcal{G}$ that yields G. Thus, the task is to decompose G recursively into subgraphs that can be generated from the nonterminals of $\mathcal{G}$. Intuitively, the NP-completeness of the problem comes from the fact that a graph has exponentially many subgraphs. This is the main difference between graph and string parsing. In the latter case, the well-known dynamic programming approach by Cocke, Kasami, and Younger is efficient because a string has only quadratically many substrings. One way to achieve polynomial parsing in the graph case as well is to make sure that only polynomially many decompositions are possible candidates for well-formed derivation trees. In this paper we achieve this by imposing restrictions on $\mathcal{G}$ which guarantee that the overall shape of a suitable decomposition of G can be "read off" G itself. Intuitively, what remains is to check whether appropriate rules of $\mathcal{G}$ can be assigned to the vertices of this decomposition in order to turn it into a derivation tree.

An attempt at a set of restrictions serving this purpose was made in (Björklund et al., 2016). Motivated by the fact that meaning representations such as those by Banarescu et al. (2013) are typically acyclic, HR grammars were considered that generate directed acyclic graphs. However, as acyclicity alone does not make parsing any easier additional conditions were placed on the form of the rules. In the present paper, we generalize the approach: the generated graphs may have cycles, the allowed rules are considerably more general, and the restrictions are fewer and formulated in an axiomatic way which allows for different concretizations. We impose two conditions on our grammars, called *reentrancy preservation* and *order preservation*. The latter is relative to an ordering of the nodes of input graphs that can be instantiated in different ways.

Let us describe the idea behind these restrictions. When working with hyperedge replacement, a nonterminal hyperedge is a placeholder attached to a sequence of of nodes. This placeholder will eventually be replaced by a subgraph that shares the attached nodes of the hyperedge (and only those) with the rest of the generated graph. One difficulty parsing has to face is that, after the replacement of a hyperedge, it may not be visible in

the resulting graph which nodes the replaced hyperedge had been attached to. Reentrancy preservation is a condition which makes it possible to recover this set of nodes from the structure of the generated graph.

One difficulty remains: even if the attached nodes of a nonterminal hyperedge can uniquely be recovered, it may still be unclear in which order they had been attached to the hyperedge. This is what is avoided by the condition of order preservation. It ensures, for example, that a rule cannot replace a nonterminal hyperedge by another nonterminal hyperedge attached to the same nodes but in a different order.

Thanks to the two restrictions, we obtain a uniform parsing algorithm which is roughly quadratic in both the size of the grammar and that of the input graph.[1]

As a final note on related work, we mention here that another recent approach to efficient parsing for HR grammars was presented in (Drewes et al., 2015, 2017), where predictive top-down and bottom-up parsers are proposed, generalizing techniques from compiler construction to the graph case. The approach thus differs from ours in that it yields a parser generator which, with only the grammar as input, constructs a quadratic parser for the specific language generated by that grammar. Provided that the grammar analysis can be performed in polynomial time (which depends on the exact variant of the parser generator used), this approach is thus uniformly polynomial as well.

The next section compiles the basic notions relevant to hyperedge replacement grammars. Section 3 and 4 define and study reentrancy and order preservation, respectively. The parsing algorithm is presented in Section 5. Section 6 presents one possible concretization of our abstract notion of preserved orders, and Section 7 concludes the paper.

2. Preliminaries

The set of non-negative integers is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$, $[n]$ denotes $\{1, \dots, n\}$. Given a set S, S^* denotes the set of all finite sequences over S, and $S^{\circledast}$ denotes the set of non-repeating sequences in S^*, i.e. those sequences in which no element of S occurs twice. The empty sequence is denoted by ε, $S^+ = S^* \setminus \{\varepsilon\}$, and $S^{\oplus} = S^{\circledast} \setminus \{\varepsilon\}$. The length of a sequence $w \in S^*$ is denoted by $|w|$, and $[w]$ denotes the smallest subset A of S such that $w \in A^*$. The canonical extensions of a mapping $f\colon S \to T$ to S^* and to the powerset of S are denoted by f as well, i.e., $f(a_1 \cdots a_k) = f(a_1) \cdots f(a_k)$ for $a_1, \dots, a_k \in S$, and $f(S') = \{f(a) \mid a \in S'\}$ for $S' \subseteq S$. A sequence $sw \in S^*$ with $s \in S$ may also be denoted by (s, w). If $\prec$ is a binary relation on S, we say that $\prec$ *orders* a given subset A of S if $A = \{s_1, \dots, s_k\}$ such that $s_1 \prec \cdots \prec s_k$, and furthermore $s_i \prec s_j$ implies $i < j$ for all $i, j \in [k]$. In this case, we denote the sequence $s_1 \cdots s_k$ (which is uniquely determined by the conditions) by $[\![A]\!]_{\prec}$. We say that a given sequence $w \in S^*$ is *ordered by* $\prec$ if $w = [\![[w]]\!]_{\prec}$.

2.1. Hypergraphs

Throughout this paper, we fix a countably infinite supply LAB of symbols called labels, such that every $\sigma \in \mathrm{LAB}$ has a unique *rank* $\mathrm{rank}(\sigma) \in \mathbb{N}$. Similarly, we fix countably infinite supplies $\mathcal{V}$ and $\mathcal{E}$ of vertices and hyperedges, respectively.

[1] The exact running time depends on how efficiently the chosen order can be computed.

Definition 2.1 (hypergraph) A (directed hyperedge-labeled) *hypergraph* over $\Sigma \subseteq \mathrm{LAB}$ is a tuple $G = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ with the following components:

- $V \subseteq \mathcal{V}$ and $E \subseteq \mathcal{E}$ are disjoint finite sets of *nodes* and *hyperedges*, respectively.
- The *attachment* $\mathrm{att}\colon E \to V^{\oplus}$ assigns to each hyperedge e a sequence of attached nodes. For $e \in E$ with $\mathrm{att}(e) = (s, t)$ we also denote s by $\mathrm{src}(e)$ and t by $\mathrm{tar}(e)$, calling them the *source* and the sequence of *targets* of e, respectively.
- The *labeling* $\mathrm{lab}\colon E \to \Sigma$ assigns a label to each hyperedge, subject to the condition that $\mathrm{rank}(\mathrm{lab}(e)) = |\mathrm{tar}(e)|$ for every $e \in E$.
- The sequence $\mathrm{ext} \in V^{\oplus}$ is the sequence of *external nodes*. If $\mathrm{ext}_G = (s, t)$, then we denote the node s by $\dot{G}$ and the sequence t of nodes by $G_{\cdot\cdot}$, respectively, and we impose the additional requirement that $\mathrm{src}(e) \notin [G_{\cdot\cdot}]$ for all $e \in E$.

The *size* $|G|$ of G is $\sum_{e \in E} |\mathrm{att}(e)|$.[2]

Note that we forbid $\mathrm{att}(e)$ (for $e \in E$) to contain any node repeatedly. In the following, we simply call hyperedges edges and hypergraphs graphs. Our division of the attachment of every edge into a single source node and any number of target nodes is similar to that used in the literature on term (hyper)graphs. It makes it meaningful to speak about directed paths (defined below). Our graphs are, however, more general than term graphs in that we, for the moment, do not impose further structural conditions on them.

Throughout the paper, if the components of a graph G are not explicitly named, we denote them by V_G, E_G, att_G, etc. If the components of G *are* given explicit names (and thus the subscript is dropped) we extend this in the obvious way to derived notations, dropping the subscript even there. We furthermore use the notation $\mathrm{out}_G(v)$ to denote the set of all outgoing edges of a node $v \in V_G$, i.e., $\mathrm{out}_G(v) = \{e \in E_G \mid \mathrm{src}_G(e) = v\}$.

An *isomorphism* $h\colon G \to H$ is a pair of bijective mappings $(h_V\colon V_G \to V_H, h_E\colon E_G \to E_H)$ such that $\mathrm{att}_H \circ h_E = h_V \circ \mathrm{att}_G$, $\mathrm{lab}_H \circ h_E = \mathrm{lab}_G$, and $\mathrm{ext}_H = h_V(\mathrm{ext}_G)$. If such an ismorphism exists we write $G \equiv H$ and say that the graphs are *isomorphic*.

A *path* of length $k \in \mathbb{N}$ from $u \in V$ to $e \in E$ in G is a sequence $p = e_1 \cdots e_k \in E^+$ where $\mathrm{src}(e_1) = u$, $\mathrm{src}(e_{i+1}) \in [\mathrm{tar}(e_i)]$ for all $i \in [k-1]$, and $e_k = e$. If furthermore v is a node in $[\mathrm{tar}(e_k)]$ then pv is a path from u to v. Both p and pv *pass* the nodes $\mathrm{src}(e_2), \ldots, \mathrm{src}(e_k)$, and we say that p *contains* $e_1, \ldots, e_k$ as well as $\mathrm{src}(e_1), \ldots, \mathrm{src}(e_k)$, while pv additionally contains v. If $\mathrm{src}(e_1) \in [\mathrm{tar}(e_k)]$, the path is a *cycle*. We say that the path is a *source path* if $u = \dot{G}$.

A node v or an edge e is *reachable* from a node u if $u = v$ or there is a path from u to v or from u to e, respectively. We simply say that v and e are *reachable in* G if they are reachable from $\dot{G}$. If G is clear from the context we may just write "reachable" instead of "reachable in G". Note that, by definition, paths are always directed, and thus all of these notions refer to directed paths.

[2] This simple definition of size is sufficient and appropriate for our purposes as the classes of grammars considered in the paper only generate connected hypergraphs, and by the definition of hypergraphs it holds that external nodes are pairwise distinct and $1 \leq |\mathrm{att}(e)| \leq |V|$ for all hyperedges e. Thus, $|V| \leq |G|$, $|E| \leq |G|$, and $|\mathrm{ext}| \leq |G|$.

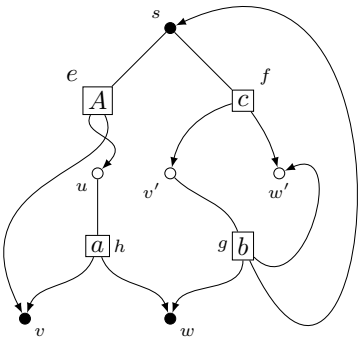

Figure 1: Example drawing of a graph G.

The *rank* of $G = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ is $\mathrm{rank}(G) = |G_{\bullet\bullet}|$ and that of $e \in E$ is $\mathrm{rank}_G(e) = \mathrm{rank}(\mathrm{lab}(e))$. The *in-degree* of a node $u \in V$ is $|\{e \in E \mid u \in [\mathrm{tar}(e)]\}|$ and its *out-degree* is $|\{e \in E \mid \mathrm{src}(e) = u\}|$. A node of out-degree 0 is a *leaf*, and a node v of in-degree 0, such that every other node in V is reachable from v, is a *root*. Thus, the root of a graph is unique if it exists. If it does, we say that G is *rooted*. Note that, if the root is $\dot{G}$, then the whole graph G is also reachable. Note furthermore that, by our general condition on the sources of edges, all nodes in $G_{\bullet\bullet}$ are leaves. The reader should keep this fact in mind because we will occasionally make use of it without explicitly mentioning it.

For a label A of rank k, we let $A^\bullet$ denote the graph $(\{0, \ldots, k\}, \{e\}, \mathrm{att}, \mathrm{lab}, 0 \cdots k)$ such that $\mathrm{att}(e) = 0 \cdots k$, and $\mathrm{lab}(e) = A$.

2.2. Drawing Conventions

We draw graphs as shown in Figure 1: external nodes are depicted as bullets and non-external ones as circles. The node $\dot{G}$ is always the topmost bullet. An edge $e \in E_G$ is depicted as a box with the edge label inscribed, which can be dropped if it is not relevant. The attachment $\mathrm{att}_G(e)$ is indicated by a line drawn from $\mathrm{src}_G(e)$ to (the box representing) e, and arrows pointing from e to the nodes in $\mathrm{tar}_G(e)$. The arrows leave the box in the order in which they appear in $\mathrm{tar}_G(e)$, from left to right. Similarly, the nodes in $G_{\bullet\bullet}$ are arranged from left to right. For example, in the figure we have $\mathrm{tar}_G(e) = uv$, $\dot{G} = s$, and $G_{\bullet\bullet} = vw$.

2.3. Hyperedge Replacement

Let H and F be graphs and $e \in E_H$ such that $V_H \cap V_F = [\mathrm{ext}_F]$, $E_H \cap E_F = \emptyset$, and $\mathrm{att}_H(e) = \mathrm{ext}_F$. The result of *substituting* e by F in H is the graph $G = H[e : F]$ such that $G = (V_H \cup V_F, (E_H \cup E_F) \setminus \{e\}, \mathrm{att}_G, \mathrm{lab}_G, \mathrm{ext}_H)$ with

$$\mathrm{att}_G(f) = \begin{cases} \mathrm{att}_H(f) & \text{if } f \in E_H \setminus \{e\} \\ \mathrm{att}_F(f) & \text{if } f \in E_F \end{cases} \qquad \mathrm{lab}_G(f) = \begin{cases} \mathrm{lab}_H(f) & \text{if } f \in E_H \setminus \{e\} \\ \mathrm{lab}_F(f) & \text{if } f \in E_F. \end{cases}$$

For graphs H and F and an edge $e \in E_H$ with $\mathrm{rank}_H(e) = \mathrm{rank}(F)$ it should be clear that we may always choose an isomorphic copy F' of F such that $H[e : F']$ is

defined. To avoid the cumbersome technicalities of constantly having to deal with explicit isomorphisms, we shall therefore always assume that F itself fulfills the requirements. If it does not, it is assumed that F is silently replaced by an appropriate isomorphic copy. Note that this is possible by our assumption that neither attachments of edges nor the sequences of external nodes of graphs contain repetitions.

For the remainder of the paper, we assume that LAB is partitioned into two disjoint subsets $\mathrm{LAB_N}$ and $\mathrm{LAB_T}$, both countably infinite, whose elements are called *nonterminals* and *terminals*, respectively. Naturally, a terminal (nonterminal) edge is an edge labeled by a terminal (nonterminal, respectively). We sometimes just call them terminals and nonterminals if there is no danger of confusion. By convention, we use capital letters to denote nonterminals, and lowercase letters for terminal symbols.

Definition 2.2 (hyperedge replacement grammar) A *hyperedge replacement grammar* (HR grammar, for short) is a system $\mathcal{G} = (\Sigma, N, S, R)$ where $\Sigma \subseteq \mathrm{LAB_T}$, $N \subseteq \mathrm{LAB_N}$, $S \in N$ is the *initial nonterminal*, and R is a set of *rules*, also called HR rules. Each rule is of the form $A \to F$ where $A \in N$ and F is a graph over $\Sigma \cup N$ with $\mathrm{rank}(F) = \mathrm{rank}(A)$.

The *size* of $\mathcal{G}$ is $|\mathcal{G}| = \sum_{(A \to F) \in R} |F|$.

For graphs G, H, we let $H \Rightarrow_R G$ if there exist a rule $A \to F \in R$ and an edge $e \in E_H$ with $\mathrm{lab}(e) = A$ such that $G = H[e : F]$. As usual, $\Rightarrow_R^*$ denotes the reflexive transitive closure of $\Rightarrow_R$. If there is no danger of confusion we often write $\Rightarrow$ and $\Rightarrow^*$ instead of $\Rightarrow_R$ and $\Rightarrow_R^*$, respectively. The *language generated by* $\mathcal{G}$ *from* $A \in \mathrm{LAB_N}$ is the set $\mathcal{L}_A(\mathcal{G})$ of all graphs G over Σ such that $A^\bullet \Rightarrow_R^* G$. The *language generated by* $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}_S(\mathcal{G})$.

For a given set $\mathcal{R}$ of HR rules (usually infinite), we let $\mathbb{G}_\mathcal{R}$ denote the set of all graphs G over LAB such that $A^\bullet \Rightarrow_\mathcal{R}^* G$ for some $A \in \mathrm{LAB_N}$.

Given pairwise distinct edges $f_1, \ldots, f_k \in E_F$ and graphs $G_1, \ldots, G_k$ such that $F[f_1 : G_1] \cdots [f_k : G_k]$ is defined, we may denote the latter by $F[f_1 : G_1, \ldots, f_k : G_k]$. We recall here the so-called context-freeness lemma of hyperedge-replacement grammars:

Lemma 2.3 (Habel (1992); Drewes et al. (1997)) Let $\mathcal{G} = (\Sigma, N, S, R)$ be an HR grammar. The sets $\mathcal{L}_A(\mathcal{G})$ $(A \in N)$ are the smallest sets such that the following holds: for every rule $(A \to F) \in R$, if $f_1, \ldots, f_k$ are the nonterminal edges in F and $G_1 \in \mathcal{L}_{\mathrm{lab}_F(f_1)}(\mathcal{G}), \ldots, G_k \in \mathcal{L}_{\mathrm{lab}_F(f_k)}(\mathcal{G})$, then $F[f_1 : G_1, \ldots, f_k : G_k]$ is in $\mathcal{L}_A(\mathcal{G})$.

3. Reentrancies

We now start to develop the notions and restrictions that lead to our parsing algorithm. This section focusses on reentrancies while the next section discusses suitable ways to order reentrant nodes.

Imagine starting at a node or edge x in a given graph G and collecting nodes that can be reached from there. Descending through G from x, we may first only encounter some nodes that cannot be reached in other ways, i.e., on source paths not containing x. However, typically we will eventually reach nodes that can *also* be reached on paths avoiding x, or are external nodes of G (which, intuitively, are nodes that can be reached from outside G). These are the reentrant nodes of x. They determine the "fringe" of a subgraph F such that $G = H[f : F]$, where H is the graph G with F "cut out" of it and f is an edge whose targets are the reentrant nodes of x in G (and thus $F_{\bullet\bullet}$ consists of the

reentrant nodes of x as well). The ambiguity inherent in this situation, caused by the fact that the reentrant nodes must be ordered in some way, will be dealt with in Section 4.

The definition below formalizes the notion of reentrant nodes.

Definition 3.1 (reentrant nodes) For a graph G and $E \subseteq E_G$, let $\mathrm{TAR}_G(E) = \bigcup_{e \in E}[\mathrm{tar}_G(e)]$ be the set of all targets of edges in E. For $x \in V_G \cup E_G$, let

$$\hat{x} = \begin{cases} x & \text{if } x \in V_G \\ \mathrm{src}_G(x) & \text{if } x \in E_G, \end{cases}$$

and let E_G^x be the set of all reachable edges $e \in E_G$ such that all source paths to e contain x. Then the set of *reentrant nodes of x in G* is

$$\mathrm{reent}_G(x) = (\mathrm{TAR}_G(E_G^x) \setminus \{\hat{x}\}) \cap (\mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G]). \tag{1}$$

Note that $e \in E_G^e$ for all reachable $e \in E_G$, and $E_G^x = \mathrm{reent}_G(x) = \emptyset$ for all unreachable x. We will not overly concern ourselves with unreachable parts of G in the following, as for the substantial parts of this paper, only graphs are of interest in which all nodes and edges are reachable.

The reentrant nodes with respect to x are those which are targets of edges that can only be reached (from $\dot{G}$) via x and are at the same time targets of other edges (not only reachable through x) or are in $[\mathrm{ext}_G]$. As indicated above, the latter corresponds to the intuition that the external nodes are those nodes which can be reached "from outside G" and thus in particular by edges not in E_G^x.

For a reachable node or edge x, we may also understand $\mathrm{reent}_G(x)$ as the nodes in G where source paths that *necessarily* pass x cross source paths that do not. Observe that, as the latter may actually be shorter than the former, a node in $\mathrm{reent}_G(x)$ may in fact be closer to $\dot{G}$ than to x itself.

Examples of reentrancies in the graph G of Figure 1 are:

1. $\mathrm{reent}_G(e) = \mathrm{reent}_G(u) = \{v, w\}$.

   This is because both of these nodes are targets of edges in $E_G^e = \{e, h\}$ and $E_G^u = \{h\}$, and they both appear in ext_G. If v and w would not be external, then w would still be in $\mathrm{reent}_G(e)$ (because it is also a target of g) but v would not. In contrast, $\mathrm{reent}_G(u)$ would remain unaffected.

2. $\mathrm{reent}_G(f) = \{w\}$.

   This is becasue $E_G^f = \{f, g\}$; here s is not reentrant because $s = \hat{f}$ and w is reentrant because it appears in ext_G (or, alternatively, because it appears in $\mathrm{tar}_G(h)$).

3. $\mathrm{reent}_G(g) = \{w, w', s\}$ because $E_G^g = \{g\}$.

4. $\mathrm{reent}_G(s) = \{v, w\}$ because $E_G^s = E_G$ and $\{v, w\} = (\mathrm{TAR}_G(E_G) \setminus \{\hat{s}\}) \cap [\mathrm{ext}_G]$.

Lemma 3.2 Let G be a graph with $x \in E_G \cup V_G$, and let $e \in E_G$ be reachable. Then $e \in E_G^x$ if and only if one of the following holds:

1. $x \in \{e, \mathrm{src}_G(e)\}$ or

2. $\mathrm{src}_G(e) \neq \dot{G}$ and all reachable edges $f \in E_G$ with $\mathrm{src}_G(e) \in [\mathrm{tar}_G(f)]$ are in E_G^x.

Proof For the *only if* direction, if $x \notin \{e, \mathrm{src}_G(e)\}$ and $\mathrm{src}_G(e) = \dot{G}$ then the source path e does not contain x and thus $e \notin E_G^x$. Thus, assume that $x \notin \{e, \mathrm{src}_G(e)\}$ and $\mathrm{src}_G(e) \neq \dot{G}$. Consider a reachable edge $f \in E_G$ with $\mathrm{src}_G(e) \in [\mathrm{tar}_G(f)]$ and assume, towards a contradiction, that $f \notin E_G^x$. Then there is a source path p to f not containing x. But then pe is a source path to e not containing x, and hence $e \notin E_G^x$.

We now prove the *if* statement. If $x \in \{e, \mathrm{src}_G(e)\}$, then all source paths to e contain x, and thus $e \in E_G^x$. Suppose now that $\mathrm{src}_G(e) \neq \dot{G}$. If all reachable edges $f \in E_G$ with $\mathrm{src}_G(e) \in [\mathrm{tar}_G(f)]$ are in E_G^x, then all source paths to e contain x as they pass one of those edges (because $\mathrm{src}_G(e) \neq \dot{G}$). Since e is reachable, it follows that $e \in E_G^x$. □

In the following, let $\approx$ be the binary relation on graphs such that $G \approx H$ if the two graphs are equal except that the order of nodes in $G_{\cdot\cdot}$ and $H_{\cdot\cdot}$ may differ. To be precise, $V_G = V_H$, $E_G = E_H$, $\mathrm{att}_G = \mathrm{att}_H$, $\mathrm{lab}_G = \mathrm{lab}_H$, $\dot{G} = \dot{H}$, and $[G_{\cdot\cdot}] = [H_{\cdot\cdot}]$. The following definition formalizes the notion of a subgraph rooted at an edge or a node. These subgraphs are uniquely determined up to $\approx$.

Definition 3.3 (rooted subgraphs) Let G be a graph and $x \in V_G \cup E_G$. The *subgraph* $G{\downarrow_x}$ *rooted at* x is a graph $H = (V, E, \mathrm{att}, \mathrm{lab}, \hat{x}w)$, where

- $E = E_G^x$ and $V = \{\hat{x}\} \cup \mathrm{TAR}_G(E)$,
- att and lab are the restrictions of att_G and lab_G to E, and
- $[w] = \mathrm{reent}_G(x)$.

Thus, $G{\downarrow_x}$ is uniquely determined up to $\approx$. We assume in the following that $G{\downarrow_x}$ denotes an arbitrarily chosen element of the corresponding equivalence class of $\approx$.[3]

A slight simplification of the definition of reentrant nodes that is easier to handle in some proofs is

$$\mathrm{ree}_G(x) = \mathrm{TAR}_G(E_G^x) \cap (\mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G]). \tag{2}$$

Obviously, $\mathrm{reent}_G(x) = \mathrm{ree}_G(x) \setminus \{\hat{x}\}$. Hence, in order to establish equations such as $\mathrm{reent}_G(x) = \mathrm{reent}_H(x)$ it is sufficient (but not necessary) to show that $\mathrm{ree}_G(x) = \mathrm{ree}_H(x)$. Thus, we will frequently show that $\mathrm{reent}_G(x) = \mathrm{reent}_H(x)$ by establishing that $\mathrm{ree}_G(x) = \mathrm{ree}_H(x)$ as the latter relieves us from considering $\hat{x}$ as a special case.

We conclude this section by stating and proving a lemma that essentially says that if y belongs to $G{\downarrow_x}$, then its rooted subgraph in G is the same as in $G{\downarrow_x}$. This will be important for the correctness proof of our parsing algorithm.

Lemma 3.4 Let G be a graph, $H = G{\downarrow_x}$ for some $x \in (V_G \cup E_G) \setminus [G_{\cdot\cdot}]$. Then $H{\downarrow_y} \approx G{\downarrow_y}$ for all $y \in (V_H \cup E_H) \setminus [\mathrm{ext}_H]$.

[3]For unreachable $x \in V_G \cup E_G$, $G{\downarrow_x}$ is the graph consisting of the single external node $\hat{x}$ and no edges.

Proof Let us first assume that x and y are both edges and that x is reachable. (As $G{\downarrow_x}$ is a single external node for unreachable x, the lemma is trivially true if x is not reachable.) By Definition 3.3 it suffices to show that

(i) $E_H^y = E_G^y$ and

(ii) $\mathrm{ree}_H(y) = \mathrm{ree}_G(y)$.

We distinguish two sub-cases.

Case 1: $x = y$. Then y is the unique edge in E_H whose source is $\dot{H}$, as all other edges (in E_G) sharing that source can obviously be reached on source paths in G not containing x.

Moreover, as all edges in E_G^x are reachable only through x, all edges in E_H are reachable in H, and all source paths (in H) pass $x = y$, meaning $E_H^y = E_H$. Consequently, $E_H^y = E_H = E_G^x = E_G^y$, completing (i).

For (ii), it suffices to note that

$$\begin{aligned}
\mathrm{ree}_H(y) &= \mathrm{TAR}_H(E_H) \cap [\mathrm{ext}_H] && \text{since } E_H^y = E_H \text{ and thus } E_H \setminus E_H^y = \emptyset \\
&= \mathrm{TAR}_G(E_G^x) \cap (\{\hat{x}\} \cup \mathrm{ree}_G(x)) && \text{by definition of } H = G{\downarrow_x} \\
&= \mathrm{ree}_G(x) \\
&= \mathrm{ree}_G(y).
\end{aligned}$$

Case 2: $x \neq y$. To prove (i), consider first an edge $e \in E_H^y \subset E_G^x$. There is a source path to y in G, and from there to e. Thus, $e \notin E_G^y$ only if e is also reachable in G on a source path not containing y. Then that path contains x (because $e \in E_G^x$), and its sub-path p from x to e cannot be a path in H because all those paths do contain y. Thus $p = p_1 e' p_2$ for some edge $e' \notin E_H = E_G^x$, i.e., e' is reachable on a source path q in G that does not contain x. However, then qep_2 is a source path to e in G, and it does not contain x, which contradicts the assumption that $e \in E_G^x$.

Conversely, for an edge $e \in E_G^y$, all source paths to e in G contain y, and hence they all contain x as well because $y \in E_G^x$. Moreover, at least one such path exists. Thus, $e \in E_G^x = E_H$. Clearly, H cannot contain more paths than G, which shows that all source paths to e in H contain y. It remains to show that at least one such path exists. However, we know that there is a source path to e in G that contains x, i.e., it has a sub-path starting at x. By the same reasoning as in the previous paragraph, this sub-path is a path in H because otherwise there would be a source path to e in G that does not contain x. Hence $e \in E_H^y$, completing the proof of (i).

We now prove (ii), i.e., $\mathrm{ree}_H(y) = \mathrm{ree}_G(y)$ (still for the case where $x, y \in E_G$ and $x \neq y$).

($\mathbf{ree_H(y) \subseteq ree_G(y)}$) We have to show that

$$\begin{aligned}
&\mathrm{TAR}_H(E_H^y) \cap (\mathrm{TAR}_H(E_H \setminus E_H^y) \cup [\mathrm{ext}_H]) \\
&\quad \subseteq \mathrm{TAR}_G(E_G^y) \cap (\mathrm{TAR}_G(E_G \setminus E_G^y) \cup [\mathrm{ext}_G]).
\end{aligned}$$

We already know that $E_H^y = E_G^y$ and hence $\mathrm{TAR}_H(E_H^y) = \mathrm{TAR}_G(E_G^y)$. Thus, it remains to be shown that $\mathrm{TAR}_H(E_H \setminus E_H^y) \cup [\mathrm{ext}_H] \subseteq \mathrm{TAR}_G(E_G \setminus E_G^y) \cup [\mathrm{ext}_G]$. Since $\mathrm{TAR}_H(E_H \setminus E_H^y) = \mathrm{TAR}_G(E_H \setminus E_G^y) \subseteq \mathrm{TAR}_G(E_G \setminus E_G^y)$, it only needs to be verified that $([\mathrm{ext}_H] \cap \mathrm{TAR}_H(E_H^y)) \setminus [\mathrm{ext}_G] \subseteq \mathrm{TAR}_G(E_G \setminus E_G^y)$, but this is clear because

$$\begin{aligned}([\mathrm{ext}_H] \cap \mathrm{TAR}_H(E_H^y)) \setminus [\mathrm{ext}_G] &= \mathrm{ree}_G(x) \setminus [\mathrm{ext}_G] \\ &\subseteq (\mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G]) \setminus [\mathrm{ext}_G] \\ &\subseteq \mathrm{TAR}_G(E_G \setminus E_G^x) \\ &\subseteq \mathrm{TAR}_G(E_G \setminus E_G^y).\end{aligned}$$

($\mathbf{ree_G(y) \subseteq ree_H(y)}$) Consider a node $v \in \mathrm{ree}_G(y)$. We already know that $v \in \mathrm{TAR}_G(E_G^y) = \mathrm{TAR}_H(E_H^y)$, so we need to verify that $v \in \mathrm{TAR}_H(E_H \setminus E_H^y) \cup [\mathrm{ext}_H]$. If $v \in [\mathrm{ext}_G]$ then there is nothing left to show, because $\mathrm{ree}_G(y) \cap [\mathrm{ext}_G] \subseteq [\mathrm{ext}_H]$. For $v \in \mathrm{ree}_G(y) \setminus [\mathrm{ext}_G]$ we get

$$\begin{aligned}&v \in \mathrm{TAR}_G(E_G^y) \cap \mathrm{TAR}_G(E_G \setminus E_G^y) \\ &\quad= \mathrm{TAR}_H(E_H^y) \cap \mathrm{TAR}_G(E_G \setminus E_H^y) \\ &\quad= \mathrm{TAR}_H(E_H^y) \cap \mathrm{TAR}_H(E_H \setminus E_H^y) \qquad\qquad (\text{since } E_G \cap E_H^y \subseteq E_H) \\ &\quad\subseteq \mathrm{TAR}_H(E_H \setminus E_H^y),\end{aligned}$$

as required.

This finishes the reasoning for the case where x, y are edges. To complete the proof, consider the case where at least one of x, y is a node. If $x = \dot{G}$, $y = \dot{G}$, or $x = y$ we obviously have $G{\downarrow_x} = G$, $H{\downarrow_y} = G{\downarrow_x}$, or $H{\downarrow_y} = H$, respectively, and there is nothing to show. Hence, assume that $\{x, y\} \cap [\mathrm{ext}_G] = \emptyset$ and $x \neq y$. Let $\overline{G}$ be the graph obtained from G by doing the following for every node $v \in V_G \setminus [G_{\bullet\bullet}]$:

- add a fresh node $\overline{v}$ and an edge e_v with $\mathrm{att}_{\overline{G}}(e_v) = v\overline{v}$ (the label of e_v does not matter), and
- for every edge $e \in E_G$ with $\mathrm{src}_G(e) = v$, define $\mathrm{src}_{\overline{G}}(e) = \overline{v}$.

The remaining components of $\overline{G}$, including the target attachments of edges, are inherited from G. The graph $\overline{H}$ is defined similarly. Now, since e_v is the unique outgoing edge of v, it holds that $\overline{G}{\downarrow_{e_v}} \approx \overline{G{\downarrow_v}}$, and similarly $\overline{H}{\downarrow_{e_v}} \approx \overline{H{\downarrow_v}}$ for nodes $v \in V_H \setminus [H_{\bullet\bullet}]$. Consequently, the first part of the proof shows that $\overline{H{\downarrow_y}} \approx \overline{G{\downarrow_y}}$. As the mapping $\overline{\ }$ is injective, this yields the result. □

4. Order-Preserving Hyperedge Replacement Grammars

Our aim in this section is to define a notion of order-preserving grammars that generalizes the type of HR grammars introduced in in Björklund et al. (2016) and also studied in Björklund et al. (2017).

The purpose of restricting HR grammars in this way is to make polynomial uniform parsing possible. We achieve this by making sure that there are partial orders on the

nodes of derivable graphs that can be computed efficiently and are compatible with hyperedge replacement in a way that can be used to guide the parsing process.

We start out with a class of HR rules that satisfy some structural requirements which make it possible to exploit the findings of the preceding section. Such rules are called *reentrancy preserving*. Next, we define the notion of a *suitable family of orders*. Finally, we define what it means for a set of HR rules to be *order preserving* for such a family of orders. The requirement is essentially that an HR replacement does not alter the relative order of any nodes, neither in the host graph nor in the right-hand side inserted into it.

Before giving the definition of reentrancy preservation, we define a type of rule that forms a special case among the reentrancy-preserving ones, the so-called duplication rule.

Definition 4.1 Consider a graph

$$F = (\{v_0, \ldots, v_n\}, \{e, e'\}, \mathrm{att}, \mathrm{lab}, v_0 v_{i_1} \cdots v_{i_k}),$$

where $\mathrm{att}(e) = v_0 \cdots v_n = \mathrm{att}(e')$, $\mathrm{lab}(e) = \mathrm{lab}(e') \in \mathrm{LAB_N}$, and $i_1 < \cdots < i_k$. If $k < n$ then F (and every graph isomorphic to F) is a *twin*, and if $k = n$ then it is a *clone*. A rule $A \to F$ is a *twin rule* if F is a twin and a *clone rule* if F is a clone with $\mathrm{lab}(e) = \mathrm{lab}(e') = A$. A *duplication rule* is either a clone or a twin rule.

Note that the right-hand side of a clone rule is uniquely determined by the left-hand side. A clone rule simply duplicates the nonterminal edge it is applied to, whereas a twin rule replaces a nonterminal edge by two "twins" having some additional targets (and, therefore, a different label).

Definition 4.2 (reentrancy-preserving rule) An HR rule $A \to F$ is *reentrancy preserving* if it is a duplication rule, or if F satisfies the following conditions:

(P1) all nodes in V_F are reachable,

(P2) the out-degree of every node is at most 1,

(P3) for every nonterminal edge e, $\mathrm{reent}_F(e) = [\mathrm{tar}_F(e)]$.

We denote the set of all reentrancy-preserving HR rules by $\mathcal{C}$, and thus the set of graphs that can be generated from $A^\bullet$ with $A \in \mathrm{LAB_N}$ using rules in $\mathcal{C}$ by $\mathbb{G}_\mathcal{C}$. Before discussing node orderings, let us study a few immediate properties of reentrancy-preserving rules and the graphs they generate.

First of all, note that all graphs $A^\bullet$ satisfy (P1)–(P3). Moreover, applying a reentrancy-preserving HR rule to a graph that satisfies (P1) and (P3) preserves these two properties. Thus, it follows by induction on the length of derivations that all graphs in $\mathbb{G}_\mathcal{C}$ satisfy (P1) and (P3) (but not necessarily (P2), owing to the existence of duplication rules).

Lemma 4.3 (reentrancy preservation) Let $G = H[e : F]$, where $H \in \mathbb{G}_\mathcal{C}$ and $(\mathrm{lab}_H(e) \to F) \in \mathcal{C}$. For all $x \in E_G \cup V_G$ we have

$$\mathrm{reent}_G(x) = \begin{cases} \mathrm{reent}_H(x) & \text{if } x \in E_H \cup V_H \\ \mathrm{reent}_F(x) & \text{if } x \in E_F \cup V_F \setminus [\mathrm{ext}_F] \end{cases}$$

We prove the two cases of Lemma 4.3 by establishing a lemma for each, i.e., Lemma 4.3 is the conjunction of Lemmas 4.4 and 4.5 proved next.

Lemma 4.4 Let $G = H[e:F]$, where $H \in \mathbb{G}_{\mathcal{C}}$ and $(\mathrm{lab}_H(e) \to F) \in \mathcal{C}$. For all $x \in (E_H \cup V_H) \setminus \{e\}$ it holds that $\mathrm{reent}_G(x) = \mathrm{reent}_H(x)$.

Proof Observe first that

$$E_G^x = \begin{cases} E_H^x \setminus \{e\} \cup E_F & \text{if } e \in E_H^x \\ E_H^x & \text{otherwise.} \end{cases} \tag{3}$$

This is because all nodes (and thus all edges) in F are reachable from $\dot{F}$ by (P1), and thus every source path in H can be converted into a source path in G by substituting a suitable source path in F for each occurrence of e, and vice versa every source path in G to $e' \in E_G$ can be converted into a source path in H to e' if $e' \neq e$ and to e otherwise, by substituting e for every maximal sub-path which is a path in F.

By the definition of hyperedge replacement, we have

$$\mathrm{TAR}_G(E_F) \cap \mathrm{TAR}_G(E_G \setminus E_F) \subseteq [\mathrm{ext}_F] = [\mathrm{att}_H(e)].$$

Thus, by equation (3), no node in $\mathrm{TAR}(E_F) \setminus [\mathrm{ext}_F]$ belongs to both $\mathrm{TAR}_G(E_G^x)$ and $(\mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G])$, i.e., to $\mathrm{ree}_G(x)$. In other words, among the nodes in V_F only the external nodes of F can be reentrant for x in G: $\mathrm{ree}_G(x) \cap V_F \subseteq [\mathrm{ext}_F]$. Only nodes in V_H could thus potentially violate the equality $\mathrm{reent}_G(x) = \mathrm{reent}_H(x)$. Hence, as $\hat{x}$ is in neither $\mathrm{reent}_G(x)$ nor $\mathrm{reent}_H(x)$, it remains to show that $v \in \mathrm{ree}_G(x) \iff v \in \mathrm{ree}_H(x)$ for all $v \in V_H \setminus \{\hat{x}\}$.

Recall that

$$\begin{aligned} \mathrm{ree}_G(x) &= \mathrm{TAR}_G(E_G^x) \cap (\mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G]) \\ \mathrm{ree}_H(x) &= \mathrm{TAR}_H(E_H^x) \cap (\mathrm{TAR}_H(E_H \setminus E_H^x) \cup [\mathrm{ext}_H]). \end{aligned}$$

Note that, by the definition of hyperedge replacement, we always have $[\mathrm{ext}_G] = [\mathrm{ext}_H]$.

We first consider the case when $e \notin E_H^x$ and thus $E_G^x = E_H^x$ and $E_G^x \cap E_F = \emptyset$. Then the left arguments of the intersections defining $\mathrm{ree}_G(x)$ and $\mathrm{ree}_H(x)$ are identical, i.e., $\mathrm{TAR}_G(E_G^x) = \mathrm{TAR}_H(E_H^x) \subseteq V_H$. Thus, since $[\mathrm{ext}_G] = [\mathrm{ext}_H]$ we need to show that $v \in \mathrm{TAR}_G(E_G \setminus E_H^x)$ if and only of $v \in \mathrm{TAR}_H(E_H \setminus E_H^x)$ for all $v \in V_H \setminus [\mathrm{ext}_H]$.

By the definition of hyperedge replacement, all edges in $E_H \setminus \{e\}$ keep their targets in G. Thus, only nodes $v \in [\mathrm{att}_H(e)]$ could potentially violate the equality, which yields two cases: either $v \in \mathrm{tar}_H(e) \setminus \mathrm{TAR}_F(E_F)$, which is prevented by (P1), or $v = \mathrm{src}_H(e)$. However, as $e \notin E_H^x$ (by assumption) and $v \notin [\mathrm{ext}_H]$, we know that $\mathrm{src}_H(e) \in \mathrm{TAR}_H(E_H \setminus E_H^x)$, as required.

We next consider the case when $e \in E_H^x$ and thus $E_G^x = E_H^x \setminus \{e\} \cup E_F$. In this case, we have $\mathrm{TAR}_H(E_H^x) \subseteq \mathrm{TAR}_G(E_G^x)$, but also $\mathrm{TAR}_H(E_H \setminus E_H^x) = \mathrm{TAR}_G(E_G \setminus E_G^x)$, due to the definition of hyperedge replacement. This makes the right arguments of the intersections defining $\mathrm{ree}_G(x)$ and $\mathrm{ree}_H(x)$ identical.

Thus, it remains to show that $v \in \mathrm{TAR}_G(E_G^x)$ if and only if $v \in \mathrm{TAR}_H(E_H^x)$ for the relevant cases. Once again, this boils down to whether there can be a node v that is a target of e but not of any edge in E_F, or a target of an edge in E_F but not of e, and the

only discrepancy that may occur is if $v = \mathrm{src}_H(e)$ and $\dot{F} \in \mathrm{TAR}_F(E_F)$. However, with $e \in E_H^x$, the only case in which $\mathrm{src}_H(e)$ may be an element of the second but not the first argument of the intersection defining $\mathrm{ree}_H(x)$ is the case $x = e$, which is excluded by the assumptions in the statement of the lemma. □

Lemma 4.5 Let $G = H[e:F]$, where $H \in \mathbb{G}_\mathcal{C}$ and $(\mathrm{lab}_H(e) \to F) \in \mathcal{C}$. For all $x \in E_F \cup V_F \setminus [\mathrm{ext}_F]$ it holds that $\mathrm{reent}_G(x) = \mathrm{reent}_F(x)$.

Proof As e is nonterminal and H belongs to $\mathbb{G}_\mathcal{C}$ and thus satisfies (P3), $\mathrm{ree}_H(e) = [\mathrm{tar}_H(e)]$. Hence, every node $v \in [\mathrm{att}_H(e)] = [\mathrm{ext}_F]$ is reachable on a source path in H not containing e, or it is in $[\mathrm{ext}_H]$. Thus, in G, v is reachable on a source path not containing any edge in E_F, or it is in $[\mathrm{ext}_G]$. In particular, $E_F^x = E_G^x$ and thus $E_G^x \cap (E_G \setminus E_F) = \emptyset$ for all $x \in E_F \cup V_F \setminus [\mathrm{ext}_F]$.

This further means that $\mathrm{TAR}_G(E_G^x) \subseteq V_F$, and as we previously established that all nodes in $[\mathrm{ext}_F]$ have source paths not passing edges in E_F or are in $[\mathrm{ext}_G]$, and are thus contained in the second argument of the intersection defining $\mathrm{ree}_G(x)$, the lemma follows from the observation that $\mathrm{att}_G(f) = \mathrm{att}_F(f)$ for all edges $f \in E_F$. □

We now formalize the notion of a suitable family of orders. These do not actually have to be orders in the mathematical sense, but are binary relations required to order the target nodes of nonterminal edges and of all right-hand sides. We thus call these relations orders to support the intuition that this is what they are used for.

Definition 4.6 (suitable family of orders) A family $\prec = (\prec_G)_{G \in \mathbb{G}_\mathcal{C}}$, where each $\prec_G$ is a binary relation on V_G, is a *suitable family of orders* if the following hold:

(S1) For all $A \in \mathrm{LAB_N}$, $A^\bullet_{\cdot\cdot}$ is ordered by $\prec_{A^\bullet}$.

(S2) For $G, G' \in \mathbb{G}_\mathcal{C}$, if $G' \equiv G$ via an isomorphism $h\colon G \to G'$ then for all $u, v \in V_G$ we have $u \prec_G v$ if and only if $h_V(u) \prec_{G'} h_V(v)$.

We are now ready to define our notion of order preservation.

Definition 4.7 (order-preserving) Let $\prec = (\prec_G)_{G \in \mathbb{G}_\mathcal{C}}$ be a suitable family of orders. A set $\mathcal{R} \subseteq \mathcal{C}$ of HR rules *preserves* $\prec$ if, for all $G = H[e:F]$ with $H \in \mathbb{G}_\mathcal{R}$, $e \in E_H$, and $(\mathrm{lab}_H(e) \to F) \in \mathcal{R}$, we have $\prec_G|_{V_H} = \prec_H$ and $\prec_G|_{V_F} = \prec_F$.

From now on, let $(\prec_G)_{G \in \mathbb{G}_\mathcal{C}}$ be a suitable family of orders which is preserved by a set $\mathcal{R} \subseteq \mathcal{C}$ of HR rules. We shall without loss of generality assume that each label in $\mathrm{LAB_N}$ occurs among the left-hand sides of rules in $\mathcal{R}$, since all other nonterminals can be removed from $\mathrm{LAB_N}$ (and the rules whose right-hand sides contain such labels can be removed from $\mathcal{R}$) without changing $\mathbb{G}_\mathcal{R}$. With this we get the following observation as a consequence of (S1) and Definition 4.7 (additionally using (S2)):

Observation 4.8 For every rule $A \to F$ in $\mathcal{R}$, $F_{\cdot\cdot}$ is ordered by $\prec_F$, and so is $\mathrm{tar}_F(e)$ for every nonterminal edge $e \in E_F$.

The first property follows from (S1) by choosing $H = A^\bullet$ in Definition 4.7, and the second follows by choosing $G = F[e:F']$ with $(\mathrm{lab}_F(e) \to F') \in \mathcal{R}$, applying the first property to F' and then using Definition 4.7 twice.

5. The Parsing Algorithm

We are now ready to develop our parsing algorithm and prove its correctness as well as analyse its running time.

5.1. Shallow Graphs

As a warm-up, we discuss how to parse when the grammar contains only duplication rules. This will provide some insights into how our general parsing algorithm handles cases where a node has more than one outgoing edge. Since duplication rules only use nonterminal labels in their right-hand sides, we only consider nonterminal labels in this section.

Definition 5.1 A graph G is *shallow* if $\dot{G}$ is its root and no path in G has length more than one. Additionally, we require G to only have nonterminal labels. A graph which is not shallow is *deep*. A rule is said to be shallow or deep depending on whether its right-hand side is.

Note that graphs containing terminal labels are always considered to be deep, even if they do not contain a path of length greater than one.

It should be clear that shallow rules only produce shallow graphs. In particular, duplication rules do. For the most part of Section 5.1, we shall restrict our attention to duplication rules. Note that, using exclusively duplication rules, every derivation of a graph G from a graph $A^\bullet$ consists of $|E_G| - 1$ steps.

Definition 5.2 A *siblinghood* in a shallow graph G is a set Sib of edges such that $\mathrm{tar}_G(e) = \mathrm{tar}_G(e')$ for all $e, e' \in Sib$. Given a siblinghood Sib we write $\mathrm{tar}_G(Sib)$ for this sequence, where $\mathrm{tar}_G(\emptyset) = G_{\bullet\bullet}$. The *size* of a siblinghood is the number of edges in it.

The next lemma gathers some useful properties of siblinghoods in graphs produced by duplication rules.

Lemma 5.3 Let R be a set of duplication rules and let $A^\bullet \Rightarrow_R^* G$ be a derivation. Then the following holds for every siblinghood Sib in G:

1. If $|E_G| > 1$ then G contains a siblinghood of size 2.
2. $G_{\bullet\bullet}$ is a subsequence of $\mathrm{tar}_G(Sib)$.
3. $\mathrm{lab}_G(e)$ is the same for all edges $e \in Sib$.
4. Let T_1 and T_2 be siblinghoods in G. Then the overlap of the targets of one of them with Sib is a subset of the overlap of the other with Sib. Formally, $[\mathrm{TAR}_G(Sib)] \cap [\mathrm{TAR}_G(T_1)]$ is a subset of $[\mathrm{TAR}_G(Sib)] \cap [\mathrm{TAR}_G(T_2)]$ or vice versa.
5. If Sib is a siblinghood of size 2, then there is a derivation of G of the form $A^\bullet \Rightarrow_R^* H \Rightarrow H[e : F]$, where $E_F = Sib$ (i.e., the very last step of the derivation introduces the siblinghood Sib).

Proof Property (1) is obvious: by the definition of siblinghoods, every duplication rule introduces a siblinghood of size 2. For the remaining properties, we proceed by induction on the length of derivations (or, equivalently, the size of E_G). To show that property (2) holds, it is clearly sufficient to show property (2′): $G_{\bullet\bullet}$ is a subsequence of $\mathrm{tar}_G(e)$ for every edge $e \in E_G$.

Let $A^\bullet = G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n = G$ be a derivation by rules in R. The base case is $G = A^\bullet$. Then G contains only one edge e, and $\mathrm{tar}_G(e) = G_{\bullet\bullet}$.

For the inductive case, we assume that the properties hold for $G_0, \ldots, G_k$. Let $G_{k+1} = G_k[e : F]$ for some edge e in G_k and a rule $A \to F$ in R with $A = \mathrm{lab}_{G_k}(e)$, where $E_F = \{f_1, f_2\}$. By the definition of duplication rules, $\mathrm{tar}_{G_k}(e)$ is a subsequence of $\mathrm{tar}_{G_{k+1}}(f_i)$, and thus property (2′) holds by the induction hypothesis.

To show properties (3)–(5), assume first that F is a clone rule and consider a siblinghood Sib. The inductive assumption that G_k satisfies property (4) immediately yields that G_{k+1} does so as well, because $\{\mathrm{tar}_{G_{k+1}}(T) \mid T \text{ is a siblinghood of } G_{k+1}\}$ is equal to $\{\mathrm{tar}_{G_k}(T) \mid T \text{ is a siblinghood of } G_k\}$, using the fact that $\mathrm{tar}_{G_{k+1}}(f_i) = \mathrm{tar}_{G_k}(e)$. We furthermore have that $\mathrm{lab}_F(f_1) = \mathrm{lab}_f(f_2) = \mathrm{lab}_{G_k}(e)$. If $Sib \cap \{f_1, f_2\} = \emptyset$, then property (3) holds by the induction hypothesis, because Sib is a siblinghood in G_k. To see that this indeed also establishes property (5) note that, if $G_k = G_{k-1}[e' : F']$ with $E_{F'} = Sib$, then $G_{k+1} = G_{k-1}[e : F][e' : F']$, i.e., the last two steps of the derivation can be interchanged. If $Sib \cap \{f_1, f_2\} = \{f_1\}$ (the other case being symmetric), then $Sib' = (Sib \setminus \{f_1\}) \cup \{e\}$ is a siblinghood in G_k, and thus again the induction hypothesis proves (3) since $\mathrm{lab}_F(f_i) = \mathrm{lab}_{G_k}(e)$. To see that property (5) holds in this case as well, suppose additionally that Sib is of size 2, say $Sib = \{f_1, f_2'\}$. Then all of f_1, f_2, and f_2' have identical attachments and labels, and we may assume by induction that $\{f_1, f_2'\}$ is introduced in G_k. However, then property (5) holds by simply applying an isomorphism that interchanges the roles of f_2 and f_2'. Finally, if $\{f_1, f_2\} \subseteq Sib$, property (3) follows by the same reasoning as before, and if Sib is additionally of size 2 and thus equal to $\{f_1, f_2\}$, then property (5) holds immediately.

To finish, assume now that F is a twin, i.e., $E_F = \{f_1, f_2\}$ and $\mathrm{lab}_F(f_1) = B = \mathrm{lab}_F(f_2)$ for a nonterminal B such that $\mathrm{rank}(B) > \mathrm{rank}(A)$. Then the only siblinghoods Sib containing f_1 or f_2 are subsets of the siblinghood $\{f_1, f_2\}$. Property (4) also holds, because $[\mathrm{TAR}_{G_{k+1}}(\{f_1, f_2\})] \setminus [\mathrm{TAR}_{G_k}(\{e\})]$ are new vertices, not connected to any edge in G_k and thus $[\mathrm{TAR}_{G_{k+1}}(T)] \cap [\mathrm{TAR}_{G_{k+1}}(Sib)]$ is equal to $[\mathrm{TAR}_{G_k}(T)] \cap [\mathrm{TAR}_{G_k}(\{e\})]$ for all siblinghoods T of G_k.[4] Property (3) clearly holds for Sib, and by the induction hypothesis also for all other siblinghoods, as those are siblinghoods in G_k. Finally, if any siblinghood Sib of G_{k+1} is of size 2, it is either equal to $\{f_1, f_2\}$, in which case property (5) holds immediately, or it is a siblinghood in G_k. In the latter case, we can again apply the induction hypothesis to the derivation of G_k and then swap the last two rule applications, thus introducing Sib in the last step, as above. □

In particular, property (4) of Lemma 5.3 implies that for every siblinghood Sib, either $\mathrm{tar}_G(Sib) = G_{\bullet\bullet}$ or there is a unique maximal subsequence $\mathrm{itar}_G(Sib)$ (for *inherited targets*) of $\mathrm{tar}_G(Sib)$ such that $\mathrm{itar}_G(Sib)$ is also a subsequence of $\mathrm{tar}_G(T)$ for some other

[4] For the sake of completeness, note additionally that $[\mathrm{TAR}_{G_{k+1}}(\emptyset)] \cap [\mathrm{TAR}_{G_{k+1}}(Sib)] = [G_{k+1\bullet\bullet}]$ by property (2). By a second application of property (2), this establishes property (4) for the case where $T_1 = \emptyset$ or $T_2 = \emptyset$.

siblinghood $T \neq Sib$. If $\mathrm{tar}_G(Sib) = G_{\bullet\bullet}$, we define $\mathrm{itar}_G(Sib)$ to be $\mathrm{tar}_G(Sib)$. If Sib is a siblinghood of G, we write $G{\downarrow_{Sib}}$ for the subgraph of G induced by Sib, where $G{\downarrow_{Sib}}_{\bullet\bullet} = \mathrm{itar}_G(Sib)$. We write $B(G{\downarrow_{Sib}})$ for the graph obtained from $G{\downarrow_{Sib}}$ by setting $\mathrm{lab}(e) = B$ for every edge e in $G{\downarrow_{Sib}}$.

In the upcoming algorithms, we use intermediate graphs that are *abstract* in the sense that each edge carries a set of labels rather than a single label:

Definition 5.4 An *abstract graph* is a tuple $G = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$, where V, E, att, and ext are as for graphs, but lab is a function from E to finite sets of labels. For a siblinghood Sib in a shallow abstract graph G, we define $\mathrm{lab}_G(Sib) = \bigcap_{e \in Sib} \mathrm{lab}_G(e)$. A *concretization* of G is a graph $H = (V_G, E_G, \mathrm{att}_G, \mathrm{lab}, \mathrm{ext}_G)$ such that $\mathrm{lab}(e) \in \mathrm{lab}_G(e)$ for all $e \in E_G$.

Given a graph G, we implicitly identify it with the abstract graph where $\mathrm{lab}(e)$ is a singleton set for each node, and vice versa.

A parsing algorithm handling the derivation of graphs by duplication rules is given as Algorithm 1. We argue its correctness below. More specifically, we argue that, given an input graph G, it returns the set of all nonterminals X such that $X^\bullet \Rightarrow^* G$.

Algorithm 1 Parsing with Duplication Rules

```
1: function shallowParse(set R of duplication rules, shallow abstract graph G)
2:     while |E_G| > 1 do
3:         if G contains no siblinghood of size 2 then
4:             return ∅
5:         choose a siblinghood Sib of size 2
6:         replace Sib in G with a new edge e with target sequence itar_G(Sib)
7:         lab(e) ← {A | ∃B ∈ lab_G(Sib): A → B(G↓_Sib)}
8:     return lab_G(e) where {e} = E_G
```

First, we note that if the condition on line 3 ever becomes true, then this is because condition 1 is violated, and thus G cannot be derived.

If this does not happen, we can view Algorithm 1 as producing a sequence $G_0, \ldots, G_n$ of abstract graphs such that $G_0 = G$ and G_n has only one edge. Let G_i and G_{i+1} be two graphs in the constructed sequence. Then there is a siblinghood Sib of size 2 in G_i and an edge e in G_{i+1} such that G_i is isomorphic to $G_{i+1}[e : G_i{\downarrow_{Sib}}]$. We argue that $\mathrm{lab}_{G_{i+1}}(e)$ is the set of all nonterminals A such that there exists a nonterminal B in $\mathrm{lab}_{G_i}(Sib)$ with $A \to B(G_i{\downarrow_{Sib}})$. To see this, suppose first that a nonterminal A was included in $\mathrm{lab}_{G_{i+1}}(e)$. Line 6 of shallowParse in the algorithm replaces the siblinghood Sib in question with a single edge e with $\mathrm{tar}_{G_{i+1}}(e) = \mathrm{itar}_{G_i}(Sib)$. On line 7, the label set of e is set to those nonterminals that have an appropriate duplication rule for some B in $\mathrm{lab}_{G_i}(Sib)$. (Note that B is equal to A if and only if the rule is a clone rule if and only if $\mathrm{tar}_{G_i}(Sib) = \mathrm{itar}_{G_i}(Sib)$.) Thus, starting with a single edge e with label A, we can use this rule to produce a twin or clone G_i'. Clearly, this graph is isomorphic to $B(G_i{\downarrow_{Sib}})$. Conversely, a similar reasoning gives us that for every $B \in \mathrm{lab}_{G_i}(Sib)$, Algorithm 1 includes a nonterminal A in $\mathrm{lab}_{G_{i+1}}(e)$ such that $A \to B(G_i{\downarrow_{Sib}})$.

We now consider the case where G_0, the input to the algorithm, is a concrete graph in the sense that every edge has a label set of size exactly one. We argue that if

Algorithm 1 eventually arrives at a graph G_n with $E_{G_n} = \{e\}$, then $\mathrm{lab}_{G_n}(e)$ is the set of all nonterminals X such that G'_0, the unique concretization of G_0 can be derived from $X^\bullet$. First, let $G_0, \ldots, G_n$ and X be as above. We argue that there are graphs $G'_n, G'_{n-1}, \ldots, G'_0$ such that

$$X^\bullet = G'_n \Rightarrow G'_{n-1} \Rightarrow \cdots \Rightarrow G'_0$$

and for each $i \in \{0, \ldots, n\}$, G'_i is a concretization of G_i. In particular, this means that G'_0 can thus be derived from $X^\bullet$.

For the induction basis, let $Sib = \{e\}$ be the unique siblinghood in G_n. Since X is in $\mathrm{lab}_{G_n}(e)$, $X^\bullet = G'_n$ is a concretization of G_n.

For the inductive case, consider G_i and G_{i+1}. As above, let e be the edge in G_{i+1} and $Sib = \{f_1, f_2\}$ the siblinghood in G_i such that $G_i = G_{i+1}[e : G_i{\downarrow_{Sib}}]$. By the induction hypothesis, there is an edge e' in G'_{i+1} such that $\mathrm{lab}(e') \in \mathrm{lab}(e)$. Let $\mathrm{lab}(e') = A$. By the reasoning above, there is a B in $\mathrm{lab}_{G_i}(Sib)$ such that $A \to B(G_i{\downarrow_{Sib}})$. This means that from G'_{i+1} we can derive a graph G'_i that is a concretization of G_i. In particular, siblinghood Sib will have label B in G'_i, which belongs to $\mathrm{lab}_{G_i}(Sib)$.

For the other direction, assume that $X^\bullet = G'_n \Rightarrow G'_{n-1} \Rightarrow \cdots \Rightarrow G'_0 = G$. By Lemma 5.3(5) we can assume that every step in the derivation introduces a siblinghood of size 2. Choosing this siblinghood on line 5 of the algorithm and proceeding by an induction similar to the one above, yields the required graphs $G_1, \ldots, G_n$.

Algorithm 1 handles only shallow graphs generated by duplication rules. However, the derivation of a shallow graph by a reentrancy-preserving grammar may also make use of shallow rules $A \to F$ where $E_F = \{f\}$ for a nonterminal edge f. Since we only consider order-preserving rules, these rules are always of the form $A \to B^\bullet$ with $\mathrm{rank}(A) = \mathrm{rank}(B)$, called chain rules in the following. Their effect is essentially the same as the effect of chain rules in context-free string grammars: they only relabel a nonterminal. Consequently, standard techniques can be used to extend Algorithm 1 appropriately, as follows. For the given HR grammar and an abstract graph G, define the (backwards) closure of G to be $cl(G) = (V_G, E_G, \mathrm{att}_G, \mathrm{lab}, \mathrm{ext}_G)$ where $\mathrm{lab}(e) = \{A \in N \mid A^\bullet \Rightarrow^* B^\bullet \text{ for some } B \in \mathrm{lab}_G(e)\}$ for all $e \in E_G$. Then the only two changes that must be made to Algorithm 1 in order to handle chain rules are:

1. On line 7 of SHALLOWPARSE, replace "$B \in \mathrm{lab}_G(Sib)$" by "$B \in \mathrm{lab}_{cl(G)}(Sib)$".
2. On line 8 of SHALLOWPARSE, replace "$\mathrm{lab}_G(e)$" by "$\mathrm{lab}_{cl(G)}(e)$".

In effect, this incorporates the application of all relabelings permitted by the grammar into the derivations constructed by Algorithm 1, in the standard way known from context-free string grammars.

Summarizing, we get the following lemma:

Lemma 5.5 SHALLOWPARSE(R, G), extended to rules of the form $A^\bullet \to B^\bullet$ as discussed above, runs in time $\mathcal{O}(|R|^2 + |G|^2)$ for every set R of shallow rules and every shallow abstract graph G, and it holds that

$$\text{SHALLOWPARSE}(R, G) = \{A \in \mathrm{LAB_N} \mid A^\bullet \Rightarrow^*_R H \text{ for some concretiazation } H \text{ of } G\}.$$

Proof The required correctness arguments were given above. To implement the generalization efficiently, note that the set of all pairs (A, B) with $A^\bullet \Rightarrow^* B^\bullet$ can be precomputed in time $|R|^2$ by a standard technique. The algorithm itself runs in time $|G|^2$ because choosing *Sib* on line 5 requires only linear time and the loop terminates after at most $|E_G|$ executions. □

The reader should note for later use that the term $|R|^2$ in the running time estimation is a one-time investment if SHALLOWPARSE(R, G) is run several times with differing G but the same set R of rules.

5.2. The general algorithm

We now present our parsing algorithm, show that it is correct, and determine its worst-case running time. Throughout this section, let $\mathcal{G} = (\Sigma, N, S, R)$ denote the order-preserving HR grammar which, together with a graph G over Σ, is the input to the parsing algorithm. Let $P \subseteq R$ be the set of deep rules in R, and $Q \subseteq R$ be the set of duplication rules (i.e., $P \cap Q = \emptyset$ and $R \setminus (P \cup Q)$ is the set of chain rules in R).

We want to parse the input graph G with respect to $\mathcal{G}$. Throughout most of the section, we will assume that $\prec = (\prec_G)_{G \in \mathbb{G}_\mathcal{C}}$ is a suitable family of orders and that $G \in \mathbb{G}_\mathcal{R}$ for a set $\mathcal{R}$ that preserves $\prec$. We generalize the reasoning to arbitrary input graphs at the end of the section. Thus, we furthermore assume that $\prec_G$ has been computed beforehand, and that G fulfills condition (P1) in Definition 4.2 (i.e., all nodes and edges are reachable on source paths), while (P3) is trivially fulfilled since G is terminal.

In the following, given a graph G and some $x \in E_G \cup V_G$ such that $\prec_G$ orders $\mathrm{reent}_G(x)$, we let $G(x)$ denote the graph such that $G(x) \approx G{\downarrow_x}$ and $\mathrm{ext}_{G(x)}$ is ordered by $\prec_G$. If $\prec_G$ does not order $\mathrm{reent}_G(x)$, then $G(x)$ is undefined. Note that, if G is given, then $G(x)$ can efficiently be represented by simply storing x and $G(x)_{\bullet\bullet}$, and in this representation it can easily be traversed in, e.g., a depth-first manner. In the following, we will thus assume that $G(e)$ is represented in this way when passed as a parameter to algorithms. Observe also that, for nodes $v \in V_G$, $G(v) = (\{v\}, \emptyset, \emptyset, \emptyset, v)$ if v is a leaf, and $G(v) = G(e)$ if v has out-degree 1 and $e \in E_G$ is the unique edge with $\mathrm{src}_G(e) = v$.

As the following lemma states, if $G \in \mathbb{G}_\mathcal{R}$ and $x \in V_G \cup E_G$, then $\mathrm{reent}_G(x)$ can be efficiently computed.

Lemma 5.6 Let $G \in \mathbb{G}_\mathcal{R}$. There is a quadratic algorithm that computes, for every $x \in V_G \cup E_G$, the set $\mathrm{reent}_G(x)$, and thus the graph $G(x)$, provided that it is defined (since we assume that $\prec_G$ has been precomputed).

Proof For every $x \in V_G \cup E_G$, we can make a depth-first search starting at $\dot{G}$ to determine the set of nodes reachable without passing x. Adding $[\mathrm{ext}_G]$, this yields $V = \mathrm{TAR}_G(E_G \setminus E_G^x) \cup [\mathrm{ext}_G]$. Afterwards, we perform a similar search along paths starting at x and not passing any node in V. This takes linear time and reveals the nodes in $\mathrm{reent}_G(x)$ as they are exactly the nodes in V which can be reached from x on such paths. Performing this procedure for all $x \in V_G \cup E_G$ thus takes quadratic time. □

Before we present the parsing algorithm, let us formulate and prove a lemma that will enable us to match deep right-hand sides against (subgraphs of) the input graph. This lemma captures one of the central reasons why order-preserving HR grammars can be parsed efficiently: the mapping of nodes in a candidate right-hand side to corresponding

nodes in the host graph is uniquely determined (if it exists). For the lemma and the rest of the section, recall from Section 2 that $\mathrm{out}_G(v)$ denotes the set of all outgoing edges of a node v in a graph G.

Lemma 5.7 Let $H = F[f_1 : G_1, \ldots, f_k : G_k]$ for graphs $F, G_1, \ldots, G_k \in \mathbb{G}_{\mathcal{R}}$, where F is a deep graph satisfying (P1)–(P3) and $f_1, \ldots, f_k$ are the nonterminal edges in F. If H is isomorphic to $G(e)$ via an isomorphism $h\colon H \to G(e)$ for a graph $G \in \mathbb{G}_{\mathcal{R}}$ and an edge $e \in E_G$, then the restriction ϕ of h_V to V_F satisfies the following, for every node $v \in V_F$:

(i) if $v = \dot{F}$, then $\phi(v) = \mathrm{src}_G(e)$;

(ii) if $v = \mathrm{src}_F(f)$ for a terminal edge $f \in E_F$, then $\mathrm{out}_G(\phi(v))$ is a singleton $\{f'\}$ with $\mathrm{lab}_G(f') = \mathrm{lab}_F(f)$ and it holds that $\phi(\mathrm{tar}_F(f)) = \mathrm{tar}_G(f')$; and

(iii) if $v = \mathrm{src}_F(f_i)$ for some $i \in [k]$, then $\phi(\mathrm{tar}_F(f_i)) = [\![\mathrm{reent}_G(v)]\!]_{\prec_G}$

Proof Consider any isomorphism h as in the statement of the lemma, let ϕ be its restriction to V_F, and $v \in V_F$.

If $v = \dot{F}$, then (i) holds by the definition of hyperedge replacement and isomorphism, because $\phi(\dot{F}) = \phi(\dot{H}) = \mathrm{src}_G(e)$.

If $v = \mathrm{src}_F(f)$ where f is terminal, since v has out-degree (at most) one, we have $\mathrm{out}_H(v) = \mathrm{out}_F(v) = \{f\}$. Hence by the definition of isomorphisms $\mathrm{out}_{G(e)}(\phi(v))$ is the singleton $\{f'\}$ with $f' = h_E(f)$, and we also have $\mathrm{lab}_G(f') = \mathrm{lab}_F(f)$. Again by the definition of hyperedge replacement and isomorphisms $\phi(\mathrm{tar}_F(f)) = \phi(\mathrm{tar}_H(f)) = \mathrm{tar}_G(f')$.

Finally, assume that $f = f_i$ for some $i \in [k]$. As F is deep and f nonterminal, it follows that $\mathrm{src}_F(f) \neq \dot{F}$. This is so because $\dot{F}$ has out-degree 1, and thus $\mathrm{src}_F(f) = \dot{F}$ implies $[\mathrm{tar}_F(f)] = \mathrm{reent}_F(f) = \mathrm{reent}_F(\dot{F}) = [F_{\bullet\bullet}]$ (where the first equality is due to (P3)), meaning that F is shallow as all nodes in $[F_{\bullet\bullet}]$ have out-degree 0. But, by the precondition of the lemma, F is not shallow.

Let $H = I[f : G_i]$ were I is the graph in $\mathbb{G}_{\mathcal{R}}$ given by

$$I = F[f_1 : G_1, \ldots, f_{i-1} : G_{i-1}, f_{i+1} : G_{i+1}, \ldots, f_k : G_k].$$

We get

$$\begin{aligned}
\mathrm{reent}_H(v) &= \mathrm{reent}_I(\mathrm{src}_F(f)) && \text{(by Lemma 4.3 and since } v = \mathrm{src}_F(f)) \\
&= \mathrm{reent}_F(\mathrm{src}_F(f)) && \text{(by } (k-1)\text{-fold application of Lemma 4.3)} \\
&= \mathrm{reent}_F(f) && \text{(because, by (P1), } \mathrm{src}_F(f) \text{ has out-degree 1)} \\
&= [\mathrm{tar}_F(f)] && \text{(by (P3))}
\end{aligned}$$

Since $\mathcal{R}$ preserves $\prec$, each of these steps furthermore preserves the order of nodes, and by Observation 4.8 $\mathrm{tar}_F(f)$ is ordered by $\prec_F$, which yields $\mathrm{tar}_F(f) = [\![\mathrm{reent}_H(v)]\!]_{\prec_H}$. As the definition of reentrancies is obviously invariant under isomorphism, and $\prec$ is so by (S2), the latter means that $\phi(\mathrm{tar}_F(f)) = [\![\mathrm{reent}_{G(e)}(\phi(v))]\!]_{\prec_{G(e)}}$.

It remains to be verified that $[\![\mathrm{reent}_{G(e)}(\phi(v))]\!]_{\prec_{G(e)}} = [\![\mathrm{reent}_G(\phi(v))]\!]_{\prec_G}$. For this, recall that F is assumed to be deep. This implies that $v \neq \dot{F}$, because $v = \dot{F}$ would imply $[\mathrm{tar}_F(f)] = \mathrm{reent}_F(f) = \mathrm{reent}_F(v) = [F_{\bullet\bullet}]$ (where the first equality holds by (P3) and the second by (P2), because $\mathrm{out}_F(v) = \{f\}$), and thus that F is shallow as all nodes in $[F_{\bullet\bullet}]$ have out-degree 0. Now, since $v \neq \dot{F}$ we have $\phi(v) \neq \mathrm{src}_G(e)$, and hence $\phi(v) \in V_{G(e)} \setminus [\mathrm{ext}_{G(e)}]$. This shows that Lemma 3.4 applies with $x = e$ and $y = \phi(v)$, telling us that $G{\downarrow_e}{\downarrow_{\phi(v)}} = G{\downarrow_{\phi(v)}}$. In particular, $\mathrm{reent}_{G(e)}(\phi(v)) = \mathrm{reent}_G(\phi(v))$ and thus $[\![\mathrm{reent}_{G(e)}(\phi(v))]\!]_{\prec_{G(e)}} = [\![\mathrm{reent}_G(\phi(v))]\!]_{\prec_G}$, as required. □

Note that ϕ in Lemma 5.7 does not depend on $G_1, \ldots, G_k$. Thus, given F, G, and e, and assuming that $\mathrm{reent}_G(v)$ has been precomputed for all $v \in V_G$, the mapping ϕ (viewed as a subset of $V_F \times V_G$) can be computed in linear time by an obvious recursion along the cases (i)–(iii). In the following, this computation is assumed to fail if it reveals an inconsistency, i.e., if $\mathrm{out}_G(\phi(v))$ is not a singleton $\{f'\}$ with $\mathrm{lab}_G(f') = \mathrm{lab}_F(f)$ in case (ii), there happen to be conflicting assignments of values to $\phi(v)$ for some v (i.e., ϕ is not a function), ϕ is not injective, or $\phi(F_{\bullet\bullet}) \neq [\![\mathrm{reent}_G(e)]\!]_{\prec_G}$.

The two main routines of the parser are $\textsc{parse}_{\mathrm{V}}(v)$ and $\textsc{parse}_{\mathrm{E}}(e)$, which are called alternately by PARSE to augment the nodes and edges of G with sets of nonterminals:

1. $\textsc{parse}_{\mathrm{V}}(v)$ augments a node $v \in V_G$ with $\mathrm{NT}(v) = \{A \in N \mid A^\bullet \Rightarrow^* G(v)\}$.
2. $\textsc{parse}_{\mathrm{E}}(e)$ augments an edge $e \in E_G$ with $\mathrm{NT}(e) = \{A \in N \mid A^\bullet \Rightarrow^* G(e)\}$.

In both cases, $\mathrm{NT}(x) = \emptyset$ if $G(x)$ is undefined.

Algorithm 2 Parsing for Order-Preserving HR Grammars

```
1:  function PARSE(order-preserving HR grammar 𝒢 = (Σ, N, S, R), graph G ∈ 𝔾_ℛ)
2:      preProcess(G)              ▷ Compute ≺_G as well as all G(x) for all x ∈ V_G ∪ E_G
3:      for x ∈ V_G ∪ E_G do
4:          if G(x) is defined then NT(x) ← ⊥
5:          else NT(x) ← ∅
6:      while NT(Ġ) = ⊥ do
7:          let x ∈ V_G ∪ E_G with NT(x) = ⊥ and
                                   NT(y) ≠ ⊥ for all y ∈ (V_{G(x)} ∪ E_{G(x)}) \ ([ext_{G(x)}] ∪ {x})
8:          if x ∈ V_G then PARSE_V(x)
9:          else PARSE_E(x)
10:     return S ∈ NT(Ġ)
```

The pseudocode, $\textsc{parse}_{\mathrm{V}}$ and $\textsc{parse}_{\mathrm{E}}$ is given as Algorithms 2, 3, and 4. $\textsc{parse}_{\mathrm{E}}$ additionally calls the subroutine MATCH, given as Algorithm 5. The following is a high-level description of how the algorithms work:

After PARSE has completed the preprocessing of the input graph, it repeatedly finds either a node or an edge x to be processed. Once it has processed x, $\mathrm{NT}(x)$ will be equal to the set of nonterminals A such that $A^\bullet \Rightarrow^* G(x)$. To be able to process x, the requirement is that $\mathrm{NT}(y)$ has already been determined for all nodes and edges y of $G(x)$ except for the external nodes and x itself. Intuitively, the algorithm proceeds bottom-up on the graph, starting by processing leaves, and moving on with any edges whose targets

are all either leaves or members of $\mathrm{reent}_G(x)$, moving on with "higher" nodes and edges only when everything "below" them has been processed. (This intuitive view should be taken with a grain of salt, because G can be cyclic.)

When parsing a node v with $\textsc{parse}_\mathrm{V}$, there are two cases:

- The node is a leaf, which is equivalent to saying that $G(v)$ is the graph $\bullet$ consisting of a single external node and no edges. In this case $G(v)$ can only be derived from a nonterminal of rank zero by applying a series of chain rules, i.e., $\mathrm{NT}(v) = \{A \in N \mid A^\bullet \Rightarrow^* \bullet\}$.
- The node has out-degree $d > 0$, which means that the derivations $A^\bullet \Rightarrow^* G(v)$ are (up to reordering derivation steps) those which begin with applying $d-1$ duplication rules yielding d nonterminal edges from which the individual graphs $G(e)$ with $\mathrm{src}_G(e) = v$ have been derived. This requires us to identify and "reverse" the expansions. In practise we construct a temporary graph G_sh that represents the relevant structures, with one edge for each of the graphs $G(e)$ such that $\mathrm{src}_G(e) = v$. We then call $\textsc{shallowParse}$ on this graph to calculate the set of possible nonterminals A such that $A^\bullet$ can yield a suitable collection of edges to derive $G(v)$ from.

$\textsc{parse}_\mathrm{E}$ mainly identifies potential matching right-hand sides to the graph $G(e)$ and uses the function $\textsc{match}$ to check their relationship in detail. In particular, we know that the rank of any nonterminal $A^\bullet$ generating $G(e)$ must match the number of reentrant nodes. In fact, since the grammar is order preserving, the ith node in $A^\bullet_{\cdot\cdot}$ corresponds to the ith node in $G(e)_{\cdot\cdot}$. Further, since $\textsc{parse}_\mathrm{V}$ handles any applications of duplication rules, $\textsc{parse}_\mathrm{E}$ does not have to deal with them.

Checking whether F matches $G(e)$ is easily done, using the mapping ϕ determined by Lemma 5.7: assuming that the sets $\mathrm{NT}(v)$ have already been computed for all internal nodes of $G(e)$, we just have to check for all nonterminal edges f of F that $\mathrm{lab}_F(f) \in \mathrm{NT}(\phi(\mathrm{src}_F(f)))$. This is done by the procedure $\textsc{match}$. Lemma 5.8 confirms formally that $\textsc{match}$ works correctly.

Lemma 5.8 Let $e \in E_G$ and assume that $\mathrm{NT}(v) = \{A \in N \mid A^\bullet \Rightarrow^* G(v)\}$ for all nodes $v \in V_{G(e)} \setminus [\mathrm{ext}_{G(e)}]$. For a rule $(A \to F) \in P$, $\textsc{match}(F, e)$ returns $\textsc{true}$ if and only if there is a derivation $F \Rightarrow^* G(e)$.

Algorithm 3 Parsing of Vertices for Order-Preserving HR Grammars

```
1: function parseV(node v)
2:     if out_G(v) = ∅ then
3:         NT(v) ← {A ∈ N | A• ⇒*_G •}          ▷ A single leaf can be derived from A•
4:     else initialize G_sh = (V, E, att, lab, ext) as the following shallow graph:
5:         E = {e ∈ E_G | src_G(e) = v}
6:         V = {v} ∪ ⋃_{e∈E} reent_G(e)
7:         ext = ext_{G(v)}
8:         lab(e) = NT(e) and att(e) = v · [[reent_G(e)]]_{≺_G} for each e ∈ E
9:         NT(v) ← shallowParse({r ∈ R | r shallow}, G_sh)
```

Algorithm 4 Parsing of Edges for Order-Preserving HR Grammars

```
1: function PARSE_E(edge e)
2:     NT ← ∅
3:     for each nonterminal A ∈ N do
4:         for each deep production A → F do
5:             if MATCH(F, e) then                    ▷ G(e) is derivable from F
6:                 NT ← NT ∪ {A}                      ▷ ... and thus from A•
7:     NT(e) ← {A ∈ N | A• ⇒* B• for some B ∈ NT}      ▷ finally, apply closure
```

Algorithm 5 Matching a deep right-hand side F against $G(e)$

```
1: function MATCH(right-hand side F, edge e ∈ E_G)
2:     compute φ according to Lemma 5.7 (return FALSE if this computation fails)
3:     if lab(f) ∈ NT(φ(src(f))) for all nonterminal edges f ∈ E_F then
4:         return TRUE
5:     else
6:         return FALSE
```

Proof Let $f_1, \dots, f_k$ be the pairwise distinct nonterminal edges of F. Assume first that MATCH(F, e) returns TRUE, and let $v_i = \phi(\mathrm{src}_F(f_i))$ and $G_i = G(e)(v_i)$ for all $i \in [k]$. By Lemma 5.7(iii), $\mathrm{ext}_{G_i} = \phi(\mathrm{att}_F(f_i))$ for all $i \in [k]$. By the definition of hyperedge replacement, this yields $G(e) \equiv F[f_1 : G_1, \dots, f_k : G_k]$, and by the test on line 3 of MATCH it holds that $\mathrm{lab}_F(f_i) \in \mathrm{NT}(v_i)$ and thus $\mathrm{lab}_F(f_i)^\bullet \Rightarrow^* G_i$ for all $i \in [k]$. Consequently, by context-freeness (Lemma 2.3) $F \Rightarrow^* F[f_1 : G_1, \dots, f_k : G_k] \equiv G(e)$, as required.

Conversely, assume that $F \Rightarrow^* G(e)$. Again by context-freeness, this means that there are graphs G_i such that $\mathrm{lab}_F(f_i)^\bullet \Rightarrow^* G_i$ (for all $i \in [k]$) and $H = F[f_1 : G_1, \dots, f_k : G_k]$ is isomorphic to $G(e)$ by some isomorphism $h \colon H \to G(e)$. In particular, $G_i \in \mathbb{G}_\mathcal{R}$, and thus Lemma 5.7 states that MATCH computes the restriction ϕ of h_V to V_F on line 2. The condition on line 3 is satisfied because $\mathrm{lab}_F(f_i)^\bullet \Rightarrow^* G_i$ and thus, by assumption, $\mathrm{lab}_F(f_i) \in \mathrm{NT}(\phi(\mathrm{src}_F(f_i)))$ for all $i \in [k]$. Hence, MATCH returns TRUE. □

We are now ready to show that PARSE (Algorithm 2) works correctly on graphs in $\mathbb{G}_\mathcal{R}$.

Lemma 5.9 A call PARSE$(\mathcal{G}, G)$ with $G \in \mathbb{G}_\mathcal{R}$ to Algorithm 2 returns true if and only if $G \in \mathcal{L}(\mathcal{G})$.

Proof We first argue that, whenever the condition on line 6 is satisfied, there exists an appropriate x on line 7 of PARSE. We use Lemma 3.4, which applies to the subgraphs $G(x)$ because $G{\downarrow_x} \approx G(x)$ (if the latter is defined). Intuitively, since Lemma 3.4 states that the subgraphs $G(x)$ are properly nested, we can always recurse down into them to find an appropriate x. For a more precise argument, choose some $x \in V_G \cup E_G$ with $\mathrm{NT}(x) = \bot$ that minimizes the cardinality of the set Y_x of all $y \in (V_{G(x)} \cup E_{G(x)}) \setminus ([\mathrm{ext}_{G(x)}] \cup \{x\})$ for which $\mathrm{NT}(y) = \bot$. We have to show that $Y_x = \emptyset$. Assume for a contradiction that there is a $y \in Y_x$. If $x \in E_G$, then $G(y) = G(x)(y)$ does not contain x (because $x \notin E^y_{G(x)}$ as it can be reached in $G(x)$ without passing y). As, furthermore, $[\mathrm{ext}_{G(x)}] \cap V_{G(y)} \subseteq [\mathrm{ext}_{G(y)}]$, this yields the contradiction that $|Y_y| < |Y_x|$. The second case is that $x \in V_G$. If y is

an edge with $\mathrm{src}_G(y) = x$, then $x \in [\mathrm{ext}_{G(y)}]$ and thus $Y_y \subseteq Y_x \setminus \{x, y\}$, again yielding a contradiction. Otherwise y is a node, or it is an edge with $\mathrm{src}_G(y) \neq x$. Hence x is either not in $V_{G(y)}$ at all, or else it belongs to $G(y)_{\bullet\bullet}$, which in both cases yields $Y_y \subseteq Y_x \setminus \{y\}$ and thus again a contradiction.

Thus, PARSE never gets stuck on line 6. To prove the statement of the lemma, we now show by induction on the number of loop executions the slightly stronger claim that $\mathrm{NT}(x) \neq \bot$ implies

$$\mathrm{NT}(x) = \begin{cases} \{A \in N \mid A^\bullet \Rightarrow^* G(x)\} & \text{if } G(x) \text{ is defined} \\ \emptyset & \text{otherwise.} \end{cases} \tag{4}$$

In particular, since $G = G(\dot{G})$, this means that $S \in \mathrm{NT}(\dot{G})$ if and only if $G \in \mathcal{L}(G)$.

The base case, before the first iteration of the loop, is given on line 5 of PARSE, i.e., $G(x)$ is undefined and $\mathrm{NT}(x) = \emptyset$. Thus, the claim holds in this case.

For the inductive case, assume now that $G(x)$ is defined and x fulfills the condition on line 7 of PARSE. We have to show that, after the execution of $\textsc{parse}_{\mathrm{V}}(x)$ (if $x \in V_G$) or $\textsc{parse}_{\mathrm{E}}(x)$ (if $x \in E_G$), it holds that $\mathrm{NT}(x) = \{A \in N \mid A^\bullet \Rightarrow^*_{\mathcal{G}} G(x)\}$.

Assume first that $x \in V_G$. If $\mathrm{out}_G(x) = \emptyset$, then $G(x)$ is the graph $\bullet$ consisting of a single node and no edge, and hence (4) reduces to $\mathrm{NT}(x) = \{A \in N \mid A^\bullet \Rightarrow^* \bullet\}$, which is ensured on line 3 of $\textsc{parse}_{\mathrm{V}}$. Otherwise, if $\mathrm{out}_G(x) = \{e_1, \ldots, e_k\}$ with $k > 0$, then the derivations $A^\bullet \Rightarrow^* G(x)$ are (by context-freeness and the induction hypothesis) exactly those which can be written in the form $A^\bullet \Rightarrow^* H \Rightarrow^* G(x) = H[f_1 : G(e_1), \ldots, f_k : G(e_k)]$, where H is a shallow graph with $E_H = \{f_1, \ldots, f_k\}$ and $\mathrm{lab}_H(f_i) \in \mathrm{NT}(e_i)$ for all $i \in [k]$. Since $\mathrm{tar}_H(f_i) = G(e_i)_{\bullet\bullet} = \mathrm{reent}_G(e_i)$ for all $i \in [k]$, the graph H is (isomorphic to) a concretization of the graph G_{sh} constructed in $\textsc{parse}_{\mathrm{V}}$. Conversely, if a concretization H of G_{sh} with $\mathrm{tar}_H(f_i) = G(e_i)_{\bullet\bullet} = \mathrm{reent}_G(e_i)$ for all $i \in [k]$ satisfies $A^\bullet \Rightarrow^* H$, then $A^\bullet \Rightarrow^* H[f_1 : G(e_1), \ldots, f_k : G(e_k)]$. Hence, Lemma 5.5 proves that $\mathrm{NT}(x) = \{A \in N \mid A^\bullet \Rightarrow^*_{\mathcal{G}} G(x)\}$ when line 9 of $\textsc{parse}_{\mathrm{V}}$ has been executed.

Finally, consider the case where $x \in E_G$. Since $G(x)$ is deep (it contains the terminal edge x), and no edge in $G(x)$ shares its source with x, all derivations $A^\bullet \Rightarrow^* G(x)$ have the form

$$A^\bullet \Rightarrow^* B^\bullet \Rightarrow F \Rightarrow^* G(x) \tag{5}$$

for some deep rule $A \to F$ in R. By the condition on line 7 of PARSE, x satisfies $\mathrm{NT}(v) = \{A \in N \mid A^\bullet \Rightarrow^* G(v)\}$ for all nodes $v \in V_{G(x)} \setminus [\mathrm{ext}_{G(x)}]$, i.e., on line 5 of $\textsc{parse}_{\mathrm{E}}$ the condition for applying Lemma 5.8 is fulfilled. Consequently, on line 7 of $\textsc{parse}_{\mathrm{E}}$, NT equals the set of all $B \in N$ such that there is a deep rule $B \to F$ with $F \Rightarrow^* G(x)$. Hence, by (5), the assignment on line 7 of $\textsc{parse}_{\mathrm{E}}$ yields $\mathrm{NT}(x) = \{A \in N \mid A^\bullet \Rightarrow^* G(x)\}$, as claimed. □

We are now ready to generalize our result to graphs that do not necessarily belong to $\mathbb{G}_{\mathcal{R}}$ and give time bounds. This yields the main result of the paper. The time bounds given are relative to the time it takes to compute $\prec_G$. In Section 6 we will see that there are suitable and natural families of orders that can efficiently be computed.

Theorem 5.10 Let $\prec$ be a suitable family of orders that is preserved by a set $\mathcal{R} \subseteq \mathcal{C}$ of HR rules, and let $f \colon \mathbb{G}_{\mathcal{R}} \to \mathbb{N}$ be a function such that both $f(G)$ and $\prec_G$ can be

computed in time $f(G)$ for every graph $G \in \mathbb{G}_{\mathcal{R}}$.[5] Then there is an algorithm which takes as input a graph G and an HR grammar $\mathcal{G} = (\Sigma, N, S, R)$ with $R \subseteq \mathcal{R}$, and decides in time $\mathcal{O}(f(G) + |G|^2 + |\mathcal{G}|^2)$ whether $G \in \mathcal{L}(\mathcal{G})$.

Proof Assume first that $G \in \mathbb{G}_{\mathcal{R}}$. By Lemma 5.9 and the definition of PARSE, $\text{PARSE}(\mathcal{G}, G)$ returns true if and only if $G \in \mathcal{L}(\mathcal{G})$. By assumption, we can compute $\prec_G$ in time $f(G)$ and by Lemma 5.6 the family $(\text{reent}_G(e))_{e \in E_G}$ can be computed in time $\mathcal{O}(|G|^2)$. As pointed out after (the proof of) Lemma 5.5, we can precompute $RELAB = \{(A, B) \in N^2 \mid A^\bullet \Rightarrow^* B^\bullet\}$ in time $|\mathcal{G}|^2$. Algorithm 5 (MATCH) runs in linear time in the size of the right-hand side F. PARSE_V and PARSE_E are called once for each node and edge, respectively. In particular, the calls of PARSE_E invoke MATCH at most $|E_G| \cdot |R|$ times in total, thus altogether taking at most $\mathcal{O}(|E_G| \cdot |\mathcal{G}|)$ computation steps if we make use of the precomputed set $RELAB$ to implement line 7 of PARSE_E efficiently. Moreover, PARSE_V runs in constant time, except for the construction of G_{sh} and the call to SHALLOWPARSE. The total size of the graphs that are passed to SHALLOWPARSE cannot exceed $|G|$ and SHALLOWPARSE itself runs in time $|G|^2$ by Lemma 5.5 (where we again make use of the precomputed set $RELAB$).[6] Hence, the total contribution is $\mathcal{O}(|G|^2)$. Altogether, this yields the upper bound $\mathcal{O}(f(G) + |G|^2 + |\mathcal{G}|^2)$ on the running time of $\text{PARSE}(\mathcal{G}, G)$ for $G \in \mathbb{G}_{\mathcal{R}}$.

Now, let us drop the assumption that $G \in \mathbb{G}_{\mathcal{R}}$. Let c be such that PARSE runs in time at most $g(\mathcal{G}, G) = c \cdot (f(G) + |G|^2 + |\mathcal{G}|^2))$ on input graphs in $\mathbb{G}_{\mathcal{R}}$. We use the old yardstick trick of computational complexity to make sure that the algorithm terminates in time $\mathcal{O}(g(\mathcal{G}, G))$: initially, $M = g(\mathcal{G}, G)$ is computed (which is possible because $f(G)$ can be computed in time $f(G)$), and a counter is initialized to zero. Then the original algorithm is executed, incrementing the counter after each computation step. If the counter reaches the value M before the algorithm terminates, the input is rejected (because this, by the correctness of the algorithm on $\mathbb{G}_{\mathcal{R}}$, proves that $G \notin \mathbb{G}_{\mathcal{R}}$).

This makes sure that the running time stays within $\mathcal{O}(g(\mathcal{G}, G)) = \mathcal{O}(f(G) + |G|^2 + |\mathcal{G}|^2)$ even for graphs not in $\mathbb{G}_{\mathcal{R}}$. However, we still have to make sure that such graphs are indeed rejected. For this, note that our parsing procedure, if it accepts G, actually determines a concrete derivation, as MATCH determines the mapping ϕ of nodes and edges of right-hand sides to those in G, i.e., it yields the isomorphic copies $A \to F'$ of rules $(A \to F) \in R$ that need to be applied in order to generate the concrete graph G. A similar mapping is obtained in SHALLOWPARSE because its parameter G_{sh}, constructed in PARSE_V, uses the actual nodes of V_G (see line 6 of PARSE_V). Thus, we can extend the algorithm in such a way that it, for all $A \in \text{NT}(e)$, stores the concrete isomorphic copy $A \to F'$ of the rule $(A \to F) \in R$ applied to e. Now, having this information, upon successful termination of PARSE it is easy to reconstruct in time $\mathcal{O}(f(G) + |G| \cdot |\mathcal{G}|)$ the concrete derivation, thereby checking that it is indeed a consistent derivation and rejecting if it is not. Upon success, we finally check the derived graph and G for equality (rather than isomorphism!). If they are indeed equal, then by definition $G \in \mathcal{L}(\mathcal{G})$. If they are not, then the correctness of PARSE on graphs in $\mathbb{G}_{\mathcal{R}}$ implies that $G \notin \mathbb{G}_{\mathcal{R}}$ and thus, in particular, $G \notin \mathcal{L}(\mathcal{G})$. □

[5] The first condition corresponds to the usual condition of time-constructibility, but here for a function that takes graphs as input, rather than for an ordinary complexity function.

[6] It may be worth noting that this is a very pessimistic estimation because the graphs G_{sh} will typically have considerably fewer edges than G.

We note here that, depending on the choice of $\mathcal{R}$ and $\prec$, it may often be possible to handle the case where the input graph is not an element of $\mathbb{G}_{\mathcal{R}}$ in another way, thus avoiding the explicit use of the yardstick argument and the construction of the derivation with the final equality check. In fact, it may be interesting to study the case where PARSE returns a positive result and thus a derivation of a graph $G' \in \mathcal{L}(\mathcal{G})$ even though $G \notin \mathcal{L}(\mathcal{G})$. This could be seen as a best effort to derive a graph G' similar to G. We will not further pursue this line of thought in the current paper, however.

6. A Suitable Family of Orders

We now present a family $\lhd$ of orders and a set $\mathcal{R} \subseteq \mathcal{C}$ of rules that preserves $\lhd$. In particular, this generalizes the order and set of rules used in Björklund et al. (2016).

Definition 6.1 (path order) Let G be a graph.

(1) For $e \in E_G$ with $\mathrm{tar}_G(e) = v_1 \cdots v_k$, we define a relation $\lhd_G^e$ on $\mathrm{reent}_G(e)$. For every $i \in [k]$, let V_i be the set of nodes in $\mathrm{reent}_G(e)$ which are reachable from v_i in $G{\downarrow_e}$ on a path not containing e. Then, for $u, v \in \mathrm{reent}_G(e)$, we let $u \lhd_G^e v$ if $u \in V_i$ and $v \in V_j$ implies $i < j$ for all $i, j \in [k]$.

(2) The *path order* $\lhd_G$ *on* V_G is defined as $\lhd_G = \bigcup_{e \in E_G} \lhd_G^e$.

Lemma 6.2 ($\lhd$ is suitable) The family $\lhd$ fulfills conditions (S1) and (S2), and is thus a suitable family of orders.

Proof For $G = A^\bullet$ with $E_G = \{e\}$ and $G_{\bullet\bullet} = v_1 \cdots v_k = \mathrm{att}_G(e)$ we have $\lhd_G = \lhd_G^e = \{(v_i, v_j) \mid 1 \leq i < j \leq k\}$ (because $V_i = \{v_i\}$ in the definition of $\lhd_G^e$). Thus, (S1) holds. Obviously, (S2) (invariance under isomorphism) also holds. □

Now, call a graph G *well formed* if it is rooted and it holds that $G_{\bullet\bullet}$ and $\mathrm{tar}_G(e)$, for every nonterminal edge $e \in E_G$, are ordered by $\lhd_G$. We let $\mathcal{R}$ be the set of all rules $A \to F$ in $\mathcal{C}$ such that F is well formed.

Example 6.3 Figure 2 shows an example of a well-formed right-hand side F with two nonterminal edges labelled A and B, respectively. Here, it is assumed that the leftmost leaf, labelled by $(1')$ in the figure, is the first node of $F_{\bullet\bullet}$ and and the other filled leaf, labelled (3), is the second. Following our general drawing conventions, the order of the outgoing tentacles of edges, from left to right, indicates the order of nodes in $\mathrm{tar}_F(e)$ for each edge e. We have

- $\lhd_F^{e_0} = \emptyset$ because the reentrant nodes $(1')$ and (3) can both be reached via the same tentacle of e_0 (i.e., via (1)),
- $(1') \lhd_F (2)$ and $(1') \lhd_F (3)$ as $(1')$ can only be reached via the first tentacle of e_1 while (2) and (3) can only be reached via its second (and the fact that the latter two *can* be reached via the same tentacle of e_1 means that $\lhd_F^{e_1}$ does *not* order them),
- $(1) \lhd_F (2) \lhd_F (3)$ because $\mathrm{reent}_F(e_3) = [\mathrm{tar}_F(e_3)] = \{(1), (2), (3)\}$, which are thus ordered according to their appearance in $\mathrm{tar}_F(e_3)$,

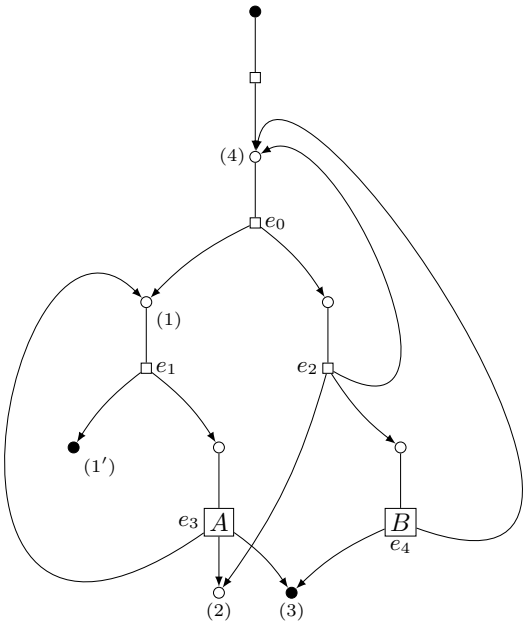

Figure 2: A well-formed right-hand side

- (2) $\lhd_F$ (3) and (2) $\lhd_F$ (4) by virtue of $\lhd_F^{e_2}$ (which does not order (3) and (4)), and finally (3) $\lhd_F$ (4) by virtue of $\lhd_F^{e_4}$.

The proof showing that $\mathcal{R}$ preserves $\lhd$ makes use of the following easy lemma.

Lemma 6.4 Let $G = H[e : F]$ for a graph H and a rule $(\mathrm{lab}_H(e) \to F) \in \mathcal{C}$, and let $u, v \in V_H$. For every path p from u to v in H there is a path p' from u to v in G containing the same nodes of H as p does. Conversely, for every path p' from u to v in G, there is a path p from u to v in G containing the same nodes of H as p' does.

Proof Since F is either a duplication rule or satisfies (P1), it contains a (simple) source path p_w to every $w \in [F_{\bullet\bullet}]$, and as all nodes in $F_{\bullet\bullet}$ have out-degree 0, this p_w does not pass any of the nodes in $F_{\bullet\bullet}$. Thus, as a path in G, p_w does not pass any node of H. It follows for the first direction that every occurrence of e in p can be replaced by a suitable p_w to obtain p'. Conversely, since $u, v \in V_H$, in p' every maximal sub-path formed by edges in E_F is of the form p_w for some w, and can thus be replaced by e to obtain p. □

Theorem 6.5 The family $\lhd$ is preserved by $\mathcal{R}$.

Proof Let $G = H[e : F]$ for $H \in \mathbb{G}_{\mathcal{R}}$, $e \in E_H$ with $\mathrm{lab}_H(e) = A$, and $(A \to F) \in \mathcal{R}$. Since $H \in \mathbb{G}_{\mathcal{R}}$, there is a derivation $A^\bullet \Rightarrow_{\mathcal{R}}^n H$ for some $n \in \mathbb{N}$. We show by induction on n that G is well formed and that, as required by Definition 4.7, $\lhd_G|_{V_H} = \lhd_H$ and $\lhd_G|_{V_F} = \lhd_F$.

As F is well formed, this holds for $n = 0$, because then $G = F$. For the induction step, we can thus assume that H is well formed. In particular, $\mathrm{tar}_H(f)$ is ordered by $\lhd_H$ (and we have $\mathrm{reent}_H(e) = [\mathrm{tar}_H(e)]$ as $H \in \mathbb{G}_{\mathcal{R}} \subseteq \mathbb{G}_{\mathcal{C}}$ satisfies (P3)).

First, let $I = V_F \setminus [\mathrm{ext}_F]$ be the set of internal nodes of F. Then, for all $v \in I$ and $f \in E_G$, $v \in \mathrm{reent}_G(f)$ only if $f \in E_F$ (by Lemma 4.3). Therefore, v appears in $\lhd^e_G$ only if $f \in E_F$. Furthermore, for $f \in E_F$, $\lhd^f_G = \lhd^f_F$ because $G{\downarrow_f} = F{\downarrow_f}$. Consequently, $\lhd_G \cap ((V_G \times I) \cup (I \times V_G)) = \lhd_F$. Moreover, since F is rooted, $\dot{F}$ occurs in none of the $\lhd^f_F$ ($f \in E_F$). As we furthermore know that $\lhd_F$ orders $F_{\cdot\cdot}$ and $\lhd_H$ orders $\mathrm{tar}_H(e)$, it remains to be shown that $\lhd^f_G = \lhd^f_H$ for all $f \in E_H \setminus \{e\}$. (Note that this also establishes well-formedness of G because H and F are well formed.)

Let $\mathrm{tar}_G(f) = v_1 \cdots v_k$ and $V = \mathrm{reent}_G(f)$. We know from Lemma 4.3 that $V = \mathrm{reent}_H(f)$. Thus, by Definition 6.1(1), it suffices to show for all $i \in [k]$ and $v \in V$ that v is reachable in $G{\downarrow_f}$ on a path p' not containing $\mathrm{src}_G(f)$ if and only if v is reachable in $H{\downarrow_f}$ on a path p not containing $\mathrm{src}_H(f)$. This, however, is true by Lemma 6.4. □

To end this section, we discuss how to compute $\lhd_G$. For this, we mainly have to explain how to compute $\lhd^e_G$ for every edge e. Below, let us say that the *inner rank* of a graph G is the maximum of the ranks of its edges and of all $|\mathrm{reent}_G(x)|$, $x \in V_G \cup E_G$. Note that if $G \in \mathcal{L}(\mathcal{G})$ for a reentrancy-preserving HR grammar $\mathcal{G} = (\Sigma, N, S, R)$, then its inner rank r is bounded by the maximal inner rank of the right-hand sides of R. Hence, in the estimations below we can always assume that r is bounded in this way because the condition can be checked during the pre-computation of all $\mathrm{reent}_G(x)$. If it is not fulfilled, the algorithm can immediately reject the input.

Lemma 6.6 For a graph G, $\lhd_G$ can be computed in time $O(r|G|^2)$, where r is the inner rank of G.

Proof It suffices to show that $\lhd^e_G$ can be computed in time $O(r|G|)$ for every edge $e \in E_G$. By Lemma 5.6, we can compute the reentrancies for all edges in quadratic time, and may thus assume that $\mathrm{reent}_G(e)$ has already been computed.

As in Definition 6.1, let $\mathrm{tar}_G(e) = v_1 \cdots v_k$ and, for every $i \in [k]$, let V_i be the set of nodes in $\mathrm{reent}_G(e)$ which are reachable from v_i in $G{\downarrow_e}$ on a path not containing e. Since all nodes in $\mathrm{reent}_G(e)$ have out-degree zero in $G{\downarrow_e}$, we can collect the V_i by simple depth-first search in $O(|G|)$ steps, and thus $O(r|G|)$ steps in total because $k \leq r$. Afterwards, we compute for each $v \in \mathrm{reent}_G(e)$ the numbers $\mathit{lower}(v) = \min\{i \in [k] \mid v \in V_i\}$ and $\mathit{upper}(v) = \max\{i \in [k] \mid v \in V_i\}$. This takes $O(r)$ steps for every v and thus $O(r^2)$ steps in total. Finally, we obtain $\lhd^e_G$ from $\mathrm{reent}_G(e) \times \mathrm{reent}_G(e)$ in $O(r^2)$ steps by deleting all pairs (u, v) such that $\mathit{upper}(u) \geq \mathit{lower}(v)$.

In summary, since $r \leq |G|$, all parts of the computation of $\lhd^e_G$ run in time $O(r|G|)$, which proves the lemma. □

From Theorem 5.10, setting $f(G) = r|G|^2$ and observing that $f(G)$ can certainly be computed in time $f(G)$, we thus get the following corollary.

Corollary 6.7 Algorithm 2, instantiated with the family $\lhd$ of orders, runs in time $O(r|G|^2 + |\mathcal{G}|^2)$, where G is the input graph, r its inner rank, and $\mathcal{G}$ the input HR grammar.

7. Conclusions and Future Work

Having developed an axiomatic notion of order-preserving hyperedge replacement grammars that allows for parsing in uniform polynomial time, and discussed a particular instantiation in the form of a suitable family of orders and a set of HR rules preserving it, several possible directions for future work remain. One is the study of suitable orders and the formal properties of these orders as well as of the sets of rules which preserve them. A better understanding of what characterizes suitable orders and order-preserving types of rules could make it easier to find additional ones. A related question is whether and in which cases it may be possible to infer a suitable order for a given set of rules "on the fly". Suppose, for example, that a given set of rules does not preserve the family $\triangleleft$ defined in Section 6. The reason may simply be that the targets of edges must be permuted depending on the labels. Is it possible to determine such a permutation efficiently if it exists? Another line of future work could extend the results of this paper to hyperedge replacement grammars with weights. Though the details may require nontrivial arguments, we believe that the parsing algorithm presented in this paper can be extended to the weighted case, so that it computes the weight of the input graph uniformly in polynomial time.

References

Aalbersberg, I. J., Ehrenfeucht, A., Rozenberg, G., 1986. On the membership problem for regular DNLC grammars. Discrete Applied Mathematics 13, 79–85.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., Schneider, N., 2013. Abstract meaning representation for sembanking. In: Proc. 7th Linguistic Annotation Workshop, ACL 2013.

Bauderon, M., Courcelle, B., 1987. Graph expressions and graph rewriting. Mathematical Systems Theory 20, 83–127.

Björklund, H., Björklund, J., Ericson, P., 2017. On the regularity and learnability of ordered DAG languages. In: Proc. 22nd International Conference on the Implementation and Application of Automata (CIAA'17). Vol. 10329 of Lecture Notes in Computer Science. Springer, pp. 27–39.

Björklund, H., Drewes, F., Ericson, P., 2016. Between a rock and a hard place – uniform parsing for hyperedge replacement DAG grammars. In: Dediu, A., Janoušek, J., Martín-Vide, C., Truthe, B. (Eds.), Proc. 10th Intl. Conf. on Language and Automata Theory and Applications. Vol. 9618 of Lecture Notes in Computer Science. pp. 521–532.

Chiang, D., Andreas, J., Bauer, D., Hermann, K. M., Jones, B., Knight, K., 2013. Parsing graphs with hyperedge replacement grammars. In: Proc. 51st Annual Meeting of the Association for Computational Linguistics (ACL 2013), Volume 1: Long Papers. pp. 924–932.

Drewes, F., 1993a. NP-completeness of k-connected hyperedge-replacement languages of order k. Information Processing Letters 45, 89–94.

Drewes, F., 1993b. Recognising k–connected hypergraphs in cubic time. Theoretical Computer Science 109, 83–122.

Drewes, F., Habel, A., Kreowski, H.-J., 1997. Hyperedge replacement graph grammars. In: Rozenberg, G. (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations. World Scientific, Ch. 2, pp. 95–162.

Drewes, F., Hoffmann, B., Minas, M., 2015. Predictive top-down parsing for hyperedge replacement grammars. In: Proc. 8th Intl. Conf. on Graph Transformation (ICGT'15). Lecture Notes in Computer Science.

Drewes, F., Hoffmann, B., Minas, M., 2017. Predictive shift-reduce parsing for hyperedge replacement grammars. In: de Lara, J., Plump, D. (Eds.), Proc. 10th Intl. Conf. on Graph Transformation (ICGT'17). Vol. 10373 of Lecture Notes in Computer Science. pp. 106–122.

Gilroy, S., Lopez, A., Maneth, S., 2017. Parsing graphs with regular graph grammars. In: Proc. 6th Joint Conf. on Lexical and Computational Semantics (*SEM 2017). pp. 199–208.

Habel, A., 1992. Hyperedge Replacement: Grammars and Languages. Vol. 643 of Lecture Notes in Computer Science. Springer.

Habel, A., Kreowski, H.-J., 1987. May we introduce to you: Hyperedge replacement. In: Proceedings of the Third Intl. Workshop on Graph Grammars and Their Application to Computer Science. Vol. 291 of Lecture Notes in Computer Science. Springer, pp. 15–26.

Lange, K.-J., Welzl, E., 1987. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. Discrete Applied Mathematics 16, 17–30.

Lautemann, C., 1990. The complexity of graph languages generated by hyperedge replacement. Acta Informatica 27, 399–421.

Vogler, W., 1991. Recognizing edge replacement graph languages in cubic time. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (Eds.), Proceedings of the Fourth Intl. Workshop on Graph Grammars and Their Application to Computer Science. Vol. 532 of Lecture Notes in Computer Science. Springer, pp. 676–687.

V

Parsing Weighted Order-Preserving Hyperedge Replacement Grammars

Henrik Björklund[1], Frank Drewes[1], and Petter Ericson[1]

Department of Computing Science, Umeå University
{henrikb,drewes,pettter}@cs.umu.se

Abstract. We introduce a weighted extension of the recently proposed notion of order-preserving hyperedge-replacement grammars and prove that the weight of a graph according to such a weighted graph grammar can be computed uniformly in quadratic time (under assumptions made precise in the paper).

1 Introduction

The hyperedge-replacement grammar (HRG) is a well-studied formalisms for describing graph languages; see, e.g., [2, 13, 12, 8]. It is also a promising candidate for modelling semantic representations of natural language such as Abstract Meaning Representation [1]. However, HRGs overshoot the mark in that parsing with respect to them is computationally too expensive. Recently, a suitable restriction called order preservation was proposed [4, 3, 5].

The present article builds upon the *order-preserving HRGs* (OPHGs) of [5]. It was shown in [5] that parsing for OPHGs is efficient, requiring polynomial time even in the uniform case, i.e. the grammar is considered to be part of the input. Here, we define a weighted version of OPHGs, and extend the results of [5] to show that when the weights are taken from a commutative semiring, we can efficiently compute the weight assigned by an OPHG to any input graph. This is an important feature since applications such as semantic modelling require ways to quantify the well-formedness of a generated graph.

Introducing weights for OPHGs requires some care, as the associativity and commutativity of some of the rules complicates the question which derivations of a certain graph are to be considered distinct. For this reason, we introduce a notion of hybrid derivation trees, in which some nodes have a set of children, while others have them ordered in a list. After this, we show how weights can efficiently be computed, and prove the correctness of the algorithm.

Related work. Another type of restricted HRGs for semantic modelling was proposed by Chiang et al. [6], together with a parsing algorithm and a detailed complexity analysis. The complexity is, however, exponential even in the non-uniform case. In particular, it is exponential in the maximum degree of nodes in the input graph. The same holds for the parsing algorithm for *regular graph grammars* presented by Gilroy et al. [11]. We also mention that another technique for efficient HRG parsing was resently developed by Drewes et al. [9, 10].

2 Preliminaries

The set of non-negative integers is $\mathbb{N}$, and $[k] = \{1, \ldots, k\}$. For a set S, S^* is the set of strings over S, while $S^{\circledast}$ is the set of strings in S^* in which no element of S occurs twice. The empty string is ϵ, and we have $S^+ = S^* \setminus \epsilon$ and $S^{\oplus} = S^{\circledast} \setminus \epsilon$. The length of a string w is denoted $|w|$. We use the terms 'string' and 'sequence' interchangably. For a sequence $w = a_1 \cdots a_n$, every sequence $a_{i_1} \cdots a_{i_k}$ with $1 \leq i_1 < \cdots < i_k \leq n$ is a *subsequence* of w, and $[w]$ is the set $\{a_1, \ldots, a_n\}$.

2.1 Hypergraphs

We fix a disjoint, countably infinite supply LAB of labels, such that each $\sigma \in \text{LAB}$ has a rank $\text{rank}(\sigma) \in \mathbb{N}$. A *hypergraph* is a structure $g = (V, E, \text{lab}, \text{att}, \text{ext})$ where V and E are the (finite) sets of *nodes* and *hyperedges*, $\text{lab} : E \to \text{LAB}$ is the *edge labelling*, $\text{att} : E \to V^{\oplus}$ is the *edge attachment* with $|\text{att}(e)| = \text{rank}(\text{lab}(e)) + 1$ for all $e \in E$, and $\text{ext} \in V^{\oplus}$ is the sequence of *external nodes*.

From now on, we simply call hypergraphs graphs, and hyperedges edges. We use the graph as a subscript to identify its components. E.g., E_g refers to the set of edges of g. For an edge $e \in E_g$ with $\text{att}(e) = v_0 \cdots v_k$, we say that $\text{src}_g(e) = v_0, \text{tar}_g(e) = v_1 \cdots v_k$, and name these the *source* and sequence of *targets*, respectively. Similarly, for $\text{ext}_g = v_0 \cdots v_l$, we say that $v_0 = \dot{g}$ is the *source* of the graph, and $v_1 \cdots v_l = g_{\cdot\cdot}$ its sequence of *targets*. In this paper, we require all targets of a graph to be leaves, i.e. $\text{src}_g(e) \notin [g_{\cdot\cdot}]$ for all $e \in E_g$. For a graph g, $\text{rank}(g) = |g_{\cdot\cdot}|$, and for an edge e, $\text{rank}(e) = \text{rank}(\text{lab}(e)) = |\text{tar}_g(e)|$. Graphs g, h are *isomorphic*, denoted $g \equiv h$, if they are equal up to a bijective renaming of nodes and edges.

For $a \in \text{LAB}$ with $\text{rank}(a) = k$, $a^{\bullet}$ denotes the graph $(\{v_0, \ldots, v_k\}, \{e\}, (e \to a), (e \to v_0 \cdots v_k), (v_0 \cdots v_k))$, i.e. the graph of one a-labelled edge of the proper rank, with all its attached nodes external.

An alternating sequence $v_1 e_1 \ldots v_k e_k$ of nodes and edges is a *path* in g from v_1 to e_k if $\text{src}_g(e_i) = v_i$ and $v_{i+1} \in [\text{tar}_g(e_i)]$, for each $i \in [k]$. We may optionally terminate the path at v_{k+1} instead of e_k. In either case, the path *passes* all nodes and edges v_i and e_i for $i \in [k]$. If $v_1 = \dot{g}$, it is a *source path*. A node v or edge e is *reachable from* s (in g) if there is a path in g from s to v (e). A node or edge is *reachable in* g if there is a source path to it.

2.2 Hyperedge replacement

For graphs h, f, and an edge $e \in E_h$ such that $\text{rank}(e) = \text{rank}(f)$, we can use *hyperedge replacement* to obtain the graph $g = h[\![e : f]\!]$, *substituting* f for e in h, where $g = ((V_h \cup V_f), (E_h \cup E_f) \setminus \{e\}, \text{att}_g, \text{lab}_g, \text{ext}_h)$ with

$$\text{att}_g(e') = \begin{cases} \text{att}_f(e') \text{ if } e' \in E_f \\ \text{att}_h(e') \text{ if } e' \in E_h \setminus \{e\} \end{cases} \text{ and } \text{lab}_g(e') = \begin{cases} \text{lab}_f(e') \text{ if } e' \in E_f \\ \text{lab}_h(e') \text{ if } e' \in E_h \setminus \{e\}. \end{cases}$$

Clearly, we can always choose isomorphic copies of h and f so that $h[\![e : f]\!]$ is defined. We will generally not make note of this, to avoid irrelevant technicalities.

For the case where $g = h[\![e : f]\!]$ and $i = g[\![e' : j]\!]$ with $e' \notin E_f$, we write $i = h[\![e : f, e' : j]\!]$, and similarly for a larger number of replacements.

We divide LAB into two subsets $\mathrm{LAB}_\mathcal{T}$ and $\mathrm{LAB}_\mathcal{N}$ of *terminals* and *nonterminals*, and accordingly call edges terminal and nonterminal ones. We sometimes shorten the expressions further to just "terminals" and "nonterminals".

2.3 Hyperedge replacement grammars

A *hyperedge replacement grammar* (HRG) is a context-free graph grammar $G = (\Sigma, N, S, R)$ consists of a *terminal alphabet* $\Sigma \subset \mathrm{LAB}_\mathcal{T}$, a *nonterminal alphabet* $N \subset \mathrm{LAB}_\mathcal{N}$, an *initial nonterminal* $S \in N$, and a set R of *(HR) rules* form $A \to f$, where $A \in N$ and f is a graph over $\Sigma \cup N$ with $\mathrm{rank}(A) = \mathrm{rank}(f)$. If f has ℓ nonterminal edges, we name them $\{e_1, \ldots, e_\ell\}$ and write $\mathit{arity}(A \to f)$ for ℓ.

If we have a graph h with an edge e with $\mathrm{lab}_h(e) = A \in N$, and $A \to f \in R$, we can *derive* $g = h[\![e : f]\!]$. We call this a *derivation step*, and denote it $h \to_{A \to f} g$. We also write more generally $h \to_G g$ for a derivation step using any rule in R. The reflexive and transitive closure of $\to_G$ is $\to_G^*$. The *language of* G is the set $\mathcal{L}(G)$ of all graphs g over $\mathrm{LAB}_\mathcal{T}$ such that $S^\bullet \to_G^* g$.

3 Order-Preserving Hyperedge Replacement Grammars

We now turn to order-preserving HRGs. The first ingredient is a condition called reentrancy preservation. Reentrancies are deeply entwined with the way we identify places in a graph that match the right-hand side of a given rule.

3.1 Reentrancies

Intuitively, the *reentrant nodes* of a node or edge x in a graph g are the first descendants of x that can also be reached on a path that avoids x. As the external nodes of a right-hand side of an HR rule are the ones that, after the replacement, are reachable from "outside" the subgraph, we also consider them as reentrant. The graph delineated by x and its reentrant nodes is the *subgraph rooted at* x.

Definition 1 (Reentrant node). *Given a graph g and $E \subset E_g$, let* $\mathrm{TAR}_g(E)$ *be the union of all sets of targets of edges in E, i.e.* $\bigcup_{e \in E} [\mathrm{tar}_g(e)]$.

Further, for $x \in V_g \cup E_g$, let $\hat{x}$ be x if $x \in V_g$, and $\mathrm{src}_g(x)$ *if $x \in E_g$. Now, let E_g^x be the set of all edges $e \in E_g$ such that all source paths to e pass x.*[1] *Then the* set of reentrant nodes of x in g *is*

$$\mathrm{reent}_g(x) = (\mathrm{TAR}_g(E_g^x) \setminus \{\hat{x}\}) \cap (\mathrm{TAR}_g(E_g \setminus E_g^x) \cup [\mathrm{ext}_g]).$$

Definition 2 (Rooted subgraph). *Given a graph g with $x \in V_g \cup E_g$, the* subgraph $g{\downarrow_x}$ rooted at x *is a graph h such that $E_h = E_g^x$, $V_h = \{\hat{x}\} \cup \mathrm{TAR}_g(E_h)$,* att_h *and* lab_h *are the appropriate restrictions of* att_g *and* lab_g*, respectively, and* ext_h *is $\hat{x}$ followed by* $\mathrm{reent}_h(x)$ *in some order.*

[1] Note that if x is not reachable in g, $E_g^x = \emptyset$

Rooted subgraphs are strictly nested, which is proved in [5] as the following lemma (where $\sim$ is isomorphy modulo the order of $g_{\bullet\bullet}$):

Lemma 1 (Lemma 3.4 in [5]). *Let g be a graph, $h = g{\downarrow_x}$ for some $x \in V_g \cup E_g$. Then $h{\downarrow_y} \sim g{\downarrow_y}$ for all $y \in (V_h \cup E_h) \setminus [\mathrm{ext}_h]$*

3.2 Reentrancy Preservation

Reentrancy preservation formalizes the property that, given a graph h and some edge $e \in E_h$ with $\mathrm{lab}_h(e)$, we can replace e by some graph f according to a rule $A \to f$ without affecting the sets $\mathrm{reent}_g(x)$ for $x \in V_h \cup V_f$.

We achieve this by restricting our grammars to two types of rules, namely *duplication rules* and *deep rules*. Rules of these two kinds are called *reentrancy preserving*. To define duplication rules, consider a graph

$$f = (\{v_0, \ldots, v_n\}, \{e_1, e_2\}, \mathrm{att}, \mathrm{lab}, \mathrm{ext}),$$

where $\mathrm{att}(e_1) = v_0 \cdots v_n = \mathrm{att}(e_2)$, $\mathrm{lab}(e_1) = \mathrm{lab}(e_2) \in \mathrm{LAB}_{\mathcal{N}}$, and ext is a subsequence of $\mathrm{att}(e_1)$ starting with v_0. If $|\mathrm{ext}| < n$ then f (and every graph isomorphic to f) is a *twin*, and if $|\mathrm{ext}| = n$ then it is a *clone*. A rule $A \to f$ is a *twin rule* if f is a twin and a *clone rule* if f is a clone with $\mathrm{lab}(e_1) = \mathrm{lab}(e_2) = A$. A *duplication rule* is either a clone or a twin rule.

A rule $A \to f$ is a *deep rule* if f fulfills the following conditions:

- $V_f \neq [\mathrm{ext}_f]$,
- all nodes in V_f are reachable from $\dot{f}$ and have out-degree ≤ 1, and
- for every nonterminal edge e, $\mathrm{reent}_f(e) = [\mathrm{tar}_f(e)]$.

A HRG is reentrancy preserving if it has only reentrancy-preserving rules. We note here that [5] also permits chain rules, i.e. rules that only change the label of an edge from one nonterminal to another nonterminal, and thus violate the first condition above. In the present paper we exclude them because they can result in an infinite number of derivations of a given graph, thus making it in general unreasonable to associate a weight with such a graph.

Later on, we will also need the following generalization of duplication rules to the case where $\ell \geq 2$ copies of a nonterminal edge are created: given any duplication rule $r = (A \to f)$ and some $\ell \geq 2$, we denote by r^ℓ the rule $A \to f'$, where f' is obtained from f by replacing e_1, e_2 by ℓ copies. Thus, $r^2 = r$.

Lemma 2 (Adapted from lemma 5.6 in [5]). *Let $g \in \mathcal{L}(G)$ for some reentrancy-preserving HRG G. There is a quadratic algorithm that computes, for every $x \in V_g \cup E_g$, the set $\mathrm{reent}_g(x)$, and thus the subgraph $g{\downarrow_x}$.*

3.3 Ordering nodes

Reentrancy preservation allows us to pinpoint the subgraphs that may have been generated by a specific nonterminal, but as shown in [4], this is not sufficient to

achieve efficient parsing, as needing to guess the order of targets in subgraphs $g{\downarrow_x}$ may still cause NP-hardness. Thus, we require a way to determine the order of nodes, in particular reentrant nodes. This requires an ordering relation that can be efficiently computed, and fulfils some basic requirements, and a set of reentrancy-preserving rules that additionally *preserves that order*. Formally:

Definition 3 (Suitable order). *For a set $\mathcal{G}$ of graphs, a* suitable family of orders *is a family $(\preceq_g)_{g\in\mathcal{G}}$ of binary relations $\preceq_g \subseteq V_g \times V_g$ such that*

- *for all $A \in \mathrm{LAB}_N$, $A^{\bullet}_{\cdot\cdot}$ is ordered by $\preceq_{A^\bullet}$ and*
- *if $i\colon g \to h$ is an isomorphism and $u, v \in V_g$, then $u \preceq_g v$ iff $i_V(u) \preceq_h i_V(v)$.*

Definition 4 (Order preservation). *A reentrancy-preserving set R of HR rules* preserves *a suitable family of orders $\preceq = (\preceq_g)_{g\in\mathcal{G}}$ if, for all $g = h[e : f]$ with $g, h, f \in \mathcal{G}$, $e \in E_h$, and $\mathrm{lab}_h(e) \to f \in R$, we have $\preceq_g|_{V_h} = \preceq_h$ and $\preceq_f|_{V_f} = \preceq_f$.*

An order-preserving HRG (OPHG) *is an HRG (Σ, N, S, R) together with a suitable family $\preceq$ of orders, such that R is both order preserving and preserves $\preceq$.*

4 Weighted Order-Preserving HR Grammars

We now add weights – taken from some semiring – to order-preserving HR grammars. For this, and throughout the rest of this paper, let $\mathcal{S} = (S, +, \cdot, 0, 1)$ be a commutative semiring, meaning that $(S, +, 0)$ and $(S, \cdot, 1)$ are two monoids over the domain S. Thus, $+$ and $\cdot$ are binary operations on S such that

- 1 is the identity element for $\cdot$
- 0 is the identity element for $+$ and the absorbing one for $\cdot$,
- $+$ and $\cdot$ are commutative, and
- $\cdot$ distributes over $+$.

As usual, for every $a \in S$ we let $a^0 = 1$ and $a^{n+1} = a \cdot a^n$ for all $n \in \mathbb{N}$.

A weighted OPHG computes a graph series, i.e. a mapping of graphs to S. As usual, this is achieved by assigning weights to rules.

Definition 5 (weighted OPHG). *A* weighted OPHG $G = (\Sigma, N, S, R, \omega)$ *(over $\mathcal{S}$) consists of an OPHG (Σ, N, S, R) and a* weight assignment $\omega\colon R \to S$.

Informally speaking, if several distinct derivations can produce the same graph, we sum up the weights of the individual derivations to obtain the weight of the graph. The weight for a single derivation is the product of the weights of all the rules applied.

It is inconvenient to formalise this based on the derivations themselves because, just as in the case of ordinary context-free grammars, derivations may differ only in the order in which nonterminals are replaced, which yields distinct derivations that should not be distinguished. A standard technique to solve this problem is to consider derivation trees instead of derivations. We can mostly use this standard

technique, but we also have to take into account that duplication rules have certain associativity and commutativity properties that make it inappropriate to sum up over derivation trees that, intuitively, should be considered equivalent.

Let us begin the process of making these notions more precise by recalling the notions of *shallow graphs* and *siblinghoods* from [5].

Definition 6. *A graph g is* shallow *if $\dot{g} = \mathrm{src}_g(e)$ for all $e \in E_g$. A* siblinghood *in g is a set* $\mathrm{Sib} \subseteq E_g$ *such that* $|\mathrm{Sib}| \geq 2$ *and* $\mathrm{tar}_g(e) = \mathrm{tar}_g(e')$ *for all* $e, e' \in \mathrm{Sib}$. *We denote* $\mathrm{tar}_g(e)$, $e \in \mathrm{Sib}$, *by* $\mathrm{tar}_g(\mathrm{Sib})$, *and let* $g(\mathrm{Sib}) = (\{\dot{g}\} \cup [\mathrm{tar}_g(\mathrm{Sib})], \mathrm{Sib}, \mathrm{att}_g|_{\mathrm{Sib}}, \mathrm{lab}_g|_{\mathrm{Sib}}, \mathrm{tar})$, *where* tar *is the subsequence of* $\mathrm{tar}_g(\mathrm{Sib})$ *of nodes that are external in g or targets of edges outside of* Sib, *i.e. that belong to* $\mathrm{TAR}_g(\mathrm{Sib}) \cap (\mathrm{TAR}_g(E_g \setminus \mathrm{Sib}) \cup [g_{\bullet\bullet}])$

For siblinghoods $\mathrm{Sib}, \mathrm{Sib}'$, *we let* $\mathrm{Sib} \leq \mathrm{Sib}'$ *if* $\mathrm{tar}_g(\mathrm{Sib})$ *is a subsequence of* $\mathrm{tar}_g(\mathrm{Sib}')$. *A siblinghood of g is* prime *if it is maximal with respect to both* $\leq$ *and set inclusion.*

From now on, we shall for technical simplicity assume that the considered OPHG G contains exactly one clone rule for every $A \in N$. This is not a restriction because the definition of the weight of derived graphs to be given below ensures that any number of clone rules for the same nonterminal can be replaced by a single clone rule whose weight is the sum of the weights of the individual rules. In particular, if there is no clone rule for A, this has the same effect as a single clone rule of weight 0. The weight of the unique clone rule for $A \in N$ is denoted by $\omega(A)$, and we write $\to_{\mathrm{cl}}$ for the derivation relation that exclusively uses clone rules, i.e. $g \to_{\mathrm{cl}}^* g'$ if g' is obtained from g by cloning nonterminal edges.

The following is essentially Lemma 5.3 of [5]:

Lemma 3. *Let $A \in N$ and let g be a shallow graph over N with $|E_g| \geq 2$.*

- *If $A^\bullet \to^+ g$, then for every prime siblinghood* Sib *of g we either have $g = g(\mathrm{Sib})$ and $A^\bullet \to_{\mathrm{cl}}^+ g$, or $A^\bullet \to^* h \to h[\![e : f]\!] \to_{\mathrm{cl}}^* h[\![e : f']\!] = g$ where $\mathrm{lab}_h(e) \to f$ is a twin rule and $g(\mathrm{Sib}) = f'$.*
- *Up to reordering of derivation steps, the derivations of these forms are the only ones deriving g from $A^\bullet$.*

Hence, a derivation of a shallow graph can be broken down into an initial series of clonings followed by iterated sub-derivations each consisting of an application of a twin rule $A \to f$ and any number of clonings of the two nonterminal edges e_1, e_2 of f. Note that the result of each such sub-derivation depends only on $A \to f$ and the number of clonings since $\mathrm{att}_f(e_1) = \mathrm{att}_f(e_2)$. Therefore, the following definition of derivation trees uses trees in which the nodes that correspond to derivations of siblinghoods are unordered and unranked. For a tree consisting of a root labelled a and subtrees $t_1, \ldots, t_\ell$, we write $a[t_1, \ldots, t_\ell]$ or $a\langle t_1, \ldots, t_\ell \rangle$ depending on whether $t_1, \ldots, t_\ell$ is to be interpreted as an ordered or unordered list (or a multiset), respectively. We write $a(t_1, \ldots, t_\ell)$ to denote a tree in which the first level of children can be either ordered or unordered.

Definition 7 (derivation tree). *For a weighted OPHG $G = (\Sigma, N, S, R, \omega)$ and $A \in N$, the set of all A-*derivation trees *is the smallest set of trees t such that one of the following holds:*

(1) $t = r[t_1, \ldots, t_\ell]$ for a deep rule $r = (A \to f) \in R$ such that $\mathit{arity}(A \to f) = \ell$, and t_i is a $\mathrm{lab}_f(e_i)$-derivation tree for every $i \in [k]$.
(2) $t = r^\ell\langle t_1, \ldots, t_\ell\rangle$ for a clone rule $A \to f$, where $\ell \geq 2$ and t_i is an A-derivation tree that is not of type (2), for every $i \in [\ell]$.
(3) $t = r^\ell\langle t_1, \ldots, t_\ell\rangle$ for a twin rule $A \to f$, where $\ell \geq 2$ and t_i is a $\mathrm{lab}_f(e_1)$-derivation tree that is not of type (2), for every $i \in [\ell]$.

A more rigorous and complete treatment of various issues surrounding derivation trees of graph algebras with associative and commutative operations can be found in [7].

We can *evaluate* a derivation tree to yield a graph g in the following way: Given a derivation tree $t = r(t_1, \ldots, t_\ell)$, $\mathit{eval}(t)$ is defined as the right-hand side f of r, with each successive nonterminal e_i replaced with the evaluation of the corresponding subtree of the derivation tree, i.e. $\mathit{eval}((A \to f)(t_1, \ldots, t_\ell)) = f[\![e_1 : \mathit{eval}(t_1), \ldots, e_\ell : \mathit{eval}(t_\ell)]\!]$. Given a graph g, we let $\mathrm{DT}_G(g)$ denote the set of all S-derivation trees such that $\mathit{eval}(t) \equiv g$.

We make the following observation, whose correctness follows from the context-freeness of hyperedge replacement.

Observation 1 For every OPHG $G = (\Sigma, N, S, R, \omega)$,

$$\mathcal{L}(G) = \{\mathit{eval}(t) \mid t \text{ is an } S\text{-derivation tree of } G\}.$$

Now, as mentioned, the weight of a graph is defined to be the sum of the weights of all its derivation trees:

Definition 8 (generated graph series). *Let $G = (\Sigma, N, S, R, \omega)$ be a weighted OPHG and $A \in N$.*

1. *For every duplication rule $r = (A \to f) \in R$ and every $\ell \geq 2$, let $\omega(r^\ell) = \omega(r) \cdot \omega(\mathrm{lab}_f(e_1))^{\ell-2}$. (Note that r^ℓ corresponds to the application of r followed by $\ell - 2$ clonings of any of the two resulting nonterminal edges.)*
2. *The weight of an A-derivation tree $t = r(t_1, \ldots, t_\ell)$ is defined inductively, as*

$$\omega(t) = \omega(r) \cdot \prod_{i \in [k]} \omega(t_i).$$

3. *The graph series $\omega_G \colon \mathcal{G}_\Sigma \to S$ generated by G is given by*

$$\omega_G(g) = \sum_{t \in \mathrm{DT}_G(g)} \omega(t).$$

(The sum is finite, and thus well defined due to the commutativity of $+$.)

Note that given G, the language $\mathcal{L}(G)$ of G seen as an unweighted grammar, is a superset of the *support* of G, i.e. the set of all graphs g such that $\omega_G(g) \neq 0$.

5 Computing Weights

Our algorithm builds upon the unweighted parsing algorithm from [5]. We store in each node and edge nothing more than an $|N|$-vector of weights, which is computed in very much the same way as the sets of nonterminals computed in [5]. We use the distributivity of multiplication over addition to keep our computations efficient (assuming efficient multiplication and addition).

The algorithm exploits Lemma 1, i.e. the property that the subgraphs $g{\downarrow_x}$ are strictly nested in all graphs derivable by an OPHL. Using this, it is possible to process the subgraphs of g in a tree-like "bottom-up" manner, marking each node and edge x with the set of all nonterminals that can generate $g{\downarrow_x}$, after all $g{\downarrow_y}$ properly contained in $g{\downarrow_x}$ have already been processed. Eventually, S belongs to the set $\dot{g}$ is marked with if and only if $g \in \mathcal{L}(G)$.

Order preservation enters the picture as follows: every subgraph h of g which was derived from some nonterminal edge, is of the form $h = g{\downarrow_x}$ for some node or edge x of g. As shown in [5], order preservation guarantees that $h_{\bullet\bullet}$ is ordered by $\preceq_g$. Thus, in the algorithm only those subgraphs $g{\downarrow_x}$ are of interest for which the ordering of targets is uniquely determined by $\preceq_g$. From now on, we will thus assume that, whenever a subgraph $h = g{\downarrow_x}$ is constructed, the order of nodes in $h_{\bullet\bullet}$ is chosen according to $\preceq_g$.

To show how $\omega_G(g)$ can be computed, we describe two algorithms in one: the first computes the derivation trees of g whereas the second computes its weight by summing up over all the derivation trees. In the current paper, we mainly use the first algorithm as a tool to facilitate the correctness proof of the second. The set of derivation trees computed can, however, be represented in a compact fashion as a "packed forest", which is of independent usefulness.

The main procedure of the algorithm computes, in the same bottom-up manner as in [5], a set $\mathcal{D}_x(A)$ of A-derivation trees for each $x \in V_g \cup E_g$ and every $A \in N$. More precisely, $\mathcal{D}_x(A)$ is the set of all A-derivation trees of the input HRG G such that $A^\bullet \to_G^* g{\downarrow_x}$. As the correctness of this procedure was proved in [5] (though not explicitly in terms of derivation trees), it remains to show that the second version of the algorithm computes $\sum_{t \in \mathcal{D}_{\dot{g}}(S)} \omega(t)$.

That second version computes weights $\mathcal{W}_x(A)$ instead of the sets $\mathcal{D}_x(A)$, where $\mathcal{W}_x(A) = \sum_{t \in \mathcal{D}_x(A))} \omega(t)$. In the pseudocode below, we always indicate the changes that must be made to obtain the second version by lines marked by "***alt:***". The corresponding line always replaces its immediate predecessor. For sets of (derivation) trees $D_1, \ldots, D_\ell$ and a rule r of arity ℓ, we furthermore write $r(D_1, \ldots, D_\ell)$ to denote the set $\{r(t_1, \ldots, t_\ell) \mid (t_1, \ldots, t_\ell) \in D_1 \times \cdots \times D_\ell\}$ (i.e. we use that notation in both the ordered and unordered case).

A subroutine used by the algorithm is Algorithm 1, a modified version of the corresponding procedure in [5]. It takes as input a shallow graph h whose edges e are already assumed to be annotated with the respective sets $\mathcal{D}_e(A)$. The algorithm uses Lemma 3 in order to assemble – in a bottom-up manner over the prime siblinghoods of h – the set $\mathcal{D}_{\dot{h}}(A)$. In the algorithm we say that a duplication rule $A \to f$ of G *fits* a siblinghood $\mathrm{Sib} = \{s_1, \ldots, s_\ell\}$ of h

Algorithm 1 Computing Derivation Trees with Duplication Rules

function SHALLOWPARSE(set R of duplication rules, shallow annotated graph h with irrelevant edge labels)
  while $|E_g| > 1$ **do**
    if h does not contain a prime siblinghood **then**
      return $(A \mapsto \emptyset)_{A \in N}$
      alt: **return** $(A \mapsto 0)_{A \in N}$
    choose a prime siblinghood $\mathrm{Sib} = \{s_1, \ldots, s_\ell\}$
    replace Sib in h by a new edge e with $\mathrm{tar}_h(e) = h(\mathrm{Sib})$
    for each $A \in N$ **do**
      $\mathcal{D}_e(A) \leftarrow \bigcup_{r = (A \to B^{\bullet\bullet}) \text{ fits Sib}} r^\ell \langle \mathcal{D}_{s_1}(B), \ldots, \mathcal{D}_{s_\ell}(B) \rangle$
      alt: $\mathcal{W}_e(A) \leftarrow \sum_{r = (A \to B^{\bullet\bullet}) \text{ fits Sib}} \omega(r^\ell) \cdot \prod_{i \in [\ell]} \mathcal{W}_{s_i}(B)$
  return $(A \mapsto \mathcal{D}_e(A))_{A \in N}$ where $\{e\} = E_h$
  alt: **return** $(A \mapsto \mathcal{W}_e(A))_{A \in N}$ where $\{e\} = E_h$

Algorithm 2 Computing Derivation Trees for Order-Preserving HR Grammars

function PARSE(order-preserving HR grammar $G = (\Sigma, N, S, R)$, graph $g \in \mathcal{G}_R$)
  preProcess(g) ▷ Compute $\prec_g$ as well as all $g{\downarrow_x}$ for all $x \in V_g \cup E_g$
  for $x \in V_g \cup E_g$ **do**
    if $g{\downarrow_x}$ is defined **then** $\mathcal{D}_x \leftarrow \bot$
    else
      $\mathcal{D}_x \leftarrow (A \mapsto \emptyset)_{A \in N}$
      alt: $\mathcal{W}_v \leftarrow (A \mapsto 0)_{A \in N}$
  while $\mathcal{D}_g = \bot$ **do**
    let $x \in V_g \cup E_g$ with $\mathcal{D}_x = \bot$ and $\mathcal{D}_y \neq \bot$ for all $y \in (V_{g{\downarrow_x}} \cup E_{g{\downarrow_x}}) \setminus ([\mathrm{ext}_{g{\downarrow_x}}] \cup \{x\})$
    if $x \in V_g$ **then** PARSE$_\mathrm{V}(x)$
    else PARSE$_\mathrm{E}(x)$
  return $\mathcal{D}_g(S)$
  alt: **return** $\mathcal{W}_g(S)$

if $f \equiv h(\{s_1, s_2\})$ when disregarding edge labels, and we denote f by $B^{\bullet\bullet}$ to indicate that the two edges in f carry the label B.

The reader should note that the result of Algorithm 1 does not depend on the choice of Sib because the prime siblinghoods $\mathrm{Sib}_1, \ldots, \mathrm{Sib}_k$ of h are pairwise disjoint and the replacement of $\mathrm{Sib} = \mathrm{Sib}_i$ by e does not affect the siblinghoods Sib_j, $j \in [k] \setminus \{i\}$ (though it may of course create an additional prime siblinghood).

The main procedure of the parsing algorithm is shown in Algorithm 2. In its while loop, it repeatedly chooses an $x \in V_g \cup E_g$ for which the sets $\mathcal{D}_x(A)$ shall be computed, and calls PARSE$_\mathrm{V}$ (Algorithm 3) or PARSE$_\mathrm{E}$ (Algorithm 4) depending on whether $x \in V_g$ or $x \in E_g$.

The function MATCHING used in line 5 of Algorithm 4 is described in [5] (using slightly different notation). It is based on the fact that, if $g{\downarrow_e}$ can be derived from a deep right-hand side f, then the mapping ϕ of the nodes in f to their

Algorithm 3 Computing Derivations Trees of $g{\downarrow_v}$ for nodes $v \in V_g$

function PARSE$_\mathrm{V}$(node v such that $\mathcal{D}_e(A) \neq \bot$ for all $e \in E_g$ with $\mathrm{src}_g(e) = v$)
  if v has out-degree 0 **then**
    $\mathcal{D}_v \leftarrow (A \mapsto \emptyset)_{A \in N}$
    alt: $\mathcal{W}_v \leftarrow (A \mapsto 0)_{A \in N}$
  else
    initialize $h = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ as the following shallow graph:
    $E = \{e \in E_g \mid \mathrm{src}_g(e) = v\}$
    $V = \{v\} \cup \bigcup_{e \in E} \mathrm{reent}_g(e)$
    $\mathrm{ext} = \mathrm{ext}_{g{\downarrow_v}}$
    $\mathrm{att}(e) = vw$, where w is $\mathrm{reent}_g(e)$ ordered by $\preceq_g$, for each $e \in E$
    $\mathcal{D}_v \leftarrow$ SHALLOWPARSE$(\{r \in R \mid r \text{ a duplication rule}\},\ h)$
    alt: $\mathcal{W}_v \leftarrow$ SHALLOWPARSE$(\{r \in R \mid r \text{ a duplication rule}\},\ h)$

Algorithm 4 Computing Derivations Trees of $g{\downarrow_e}$ for edges $e \in E_g$

function PARSE$_\mathrm{E}$(edge e s.t. $\mathcal{D}_y \neq \bot$ for all $y \in (V_{g(x)} \cup E_{g(x)}) \setminus ([\mathrm{ext}_{g(x)}] \cup \{x\})$)
  $\mathcal{D}_e(A) \leftarrow \emptyset$ for all $A \in N$
  alt: $\mathcal{W}_e(A) \leftarrow 0$ for all $A \in N$
  for each deep rule $r = (A \to f)$ of arity ℓ **do**
    $\phi \leftarrow$ MATCHING(f, e)
    if $\phi \neq$ null **then**
      $\mathcal{D}_e(A) \leftarrow \mathcal{D}_e(A) \cup r[\mathcal{D}_{\phi(\mathrm{src}_f(e_1))}(\mathrm{lab}_f(e_1)), \ldots, \mathcal{D}_{\phi(\mathrm{src}_f(e_\ell))}(\mathrm{lab}_f(e_\ell))]$
      alt: $\mathcal{W}_e(A) \leftarrow \mathcal{W}_e(A) + \omega(r) \cdot \prod_{i \in [\ell]} \mathcal{W}_{\phi(\mathrm{src}_f(e_i))}(\mathrm{lab}_f(e_i))\}$

images in $g{\downarrow_e}$ is uniquely determined by f and the reentrancies in $g{\downarrow_e}$, due to reentrancy and order preservation. As proved in [5], this makes it furthermore possible to compute $\phi = \text{MATCHING}(f, e)$ in linear time.

As the correctness of the computation of the sets $\mathcal{D}_x(A)$ was essentially shown in [5], we take it for granted here and use it to show inductively that the weights are correctly computed. Below, we assume for the sake of technical simplicity that the operations of the semiring $\mathcal{S}$ are computable in constant time.

Theorem 2. *Let $\prec$ be a suitable family of orders, and let η be a function mapping graphs to $\mathbb{N}$ such that both $\eta(g)$ and $\prec_g$ can be computed in time $\eta(g)$.*[2] *Then there is an algorithm which takes as input a graph g and an OPHG grammar $G = (\Sigma, N, S, R, \omega)$, and computes $\omega_G(g)$ in time $\mathcal{O}(\eta(g) + |g|^2 + |G|^2)$.*

Proof. With straightforward reformulations, the proof of the main theorem in [5] shows that Algorithm 2 computes $\mathrm{DT}_G(g)$ and runs in time $\mathcal{O}(\eta(g) + |g|^2 + |G|^2)$ if the time required for the explicit construction of derivation trees is neglected.[3] Together with the assumption that the operations of $\mathcal{S}$ can be computed in

[2] The function η describes the complexity of computing $\prec_g$, and the condition that it can be executed in time $\eta(g)$ corresponds to the usual requirement of time constructibility.

[3] Instead of computing the sets $\mathcal{D}_x(A)$, the algorithm in [5] only computes, for every $x \in V_g \cup E_g$, the set of all $A \in N$ such that $\mathcal{D}_x(A) \neq \emptyset$.

constant time, the latter means that the weight-computing version of Algorithm 2 runs in time $\mathcal{O}(\eta(g) + |g|^2 + |G|^2)$ as well. To complete the proof, it thus suffices to prove by induction that Algorithms 1–4 maintain the invariant that $\mathcal{W}_x(A) = \sum_{t \in \mathcal{D}_x(A)} \omega(t)$ for those edges and nodes x and those $A \in N$ such that $\mathcal{D}_x(A) \neq \bot$.

In the proof, for a set D of derivation trees, we abbreviate $\sum_{t \in D} \omega(t)$ by $\omega(D)$. We check the algorithms one by one. Note that the induction hypothesis states that the equation $\mathcal{W}_x(A) = \omega(\mathcal{D}_x(A))$ holds when the respective procedure is entered, and we have to show that it still holds afterwards. We use the fact that, by distributivity, for every rule $r = (A \to f)$ of arity ℓ and all sets $D_1, \ldots, D_\ell$ of derivation trees, it holds that

$$\omega(r(D_1, \ldots, D_\ell)) = \omega(r) \cdot \prod_{i \in [\ell]} \omega(D_i). \tag{1}$$

Procedure SHALLOWPARSE*:* We have to show that the two lines in the body of the loop starting in line 8 maintain the invariant. These lines change only $\mathcal{D}_e(A)$ and $\mathcal{W}_e(A)$, and after those two lines we have

$$\begin{aligned}
\mathcal{W}_e(A) &= \sum_{r = (A \to B^{\bullet\bullet}) \text{ fits Sib}} \omega(r^\ell) \cdot \prod_{i \in [\ell]} \mathcal{W}_{s_i}(B) \\
&= \sum_{r = (A \to B^{\bullet\bullet}) \text{ fits Sib}} \omega(r^\ell) \cdot \prod_{i \in [\ell]} \omega(\mathcal{D}_{s_i}(B)) \\
&= \sum_{r = (A \to B^{\bullet\bullet}) \text{ fits Sib}} \omega(r^\ell \langle \mathcal{D}_{s_1}(B), \ldots, \mathcal{D}_{s_\ell}(B) \rangle) \quad \text{(by Equation 1)} \\
&= \omega(\mathcal{D}_e(A)).
\end{aligned}$$

Procedure PARSE*:* Only lines 6 and 7 affect some $\mathcal{D}_x(A)$ and $\mathcal{W}_x(A)$. These lines obviously preserve the invariant.

Procedure PARSE$_{\text{V}}$*:* As before, lines 3 and 4 respect the invariant. Concerning lines 11 and 12, note that the two versions of SHALLOWPARSE return $(A \mapsto \mathcal{D}_e(A))_{A \in N}$ and $(A \mapsto \mathcal{W}_e(A))_{A \in N}$, respectively, for some edge e. By induction hypothesis, $\mathcal{W}_e(A) = \omega(\mathcal{D}_e(A))$ for all $A \in N$, which completes the argument.

Procedure PARSE$_{\text{E}}$*:* Once more, lines 2 and 3 respect the invariant. Furthermore, if $D = \mathcal{D}_e(A)$ and $W = \mathcal{W}_e(A) = \omega(\mathcal{D}_e(A))$ before an execution of lines 7 and 8 then, after those two lines,

$$\begin{aligned}
\mathcal{W}_e(A) &= W + \omega(r) \cdot \prod_{i \in [\ell]} \mathcal{W}_{\phi(\mathrm{src}_f(e_i))}(\mathrm{lab}_f(e_i))\} \\
&= \omega(D) + \omega(r) \cdot \prod_{i \in [\ell]} \omega(\mathcal{D}_{\phi(\mathrm{src}_f(e_i))}(\mathrm{lab}_f(e_i))) \\
&= \omega(D) + \omega(r[\mathcal{D}_{\phi(\mathrm{src}_f(e_1))}(\mathrm{lab}_f(e_1)), \ldots, \mathcal{D}_{\phi(\mathrm{src}_f(e_\ell))}(\mathrm{lab}_f(e_\ell))] \\
&= \omega(\mathcal{D}_e(A)).
\end{aligned}$$

This completes the correctness proof of the theorem. □

As indicated before, it is worthwhile noticing that the first version of the parsing algorithm computes the set $\mathrm{DT}_G(g)$ in time $\mathcal{O}(\eta(g) + |g|^2 + |G|^2)$ if the sets $\mathcal{D}_x(A)$ are represented in a compact way as packed forests. This may be useful for further applications.

References

1. Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., Schneider, N.: Abstract meaning representation for sembanking. In: Proc. 7th Linguistic Annotation Workshop, ACL 2013 (2013)
2. Bauderon, M., Courcelle, B.: Graph expressions and graph rewriting. Mathematical Systems Theory **20**, 83–127 (1987)
3. Björklund, H., Björklund, J., Ericson, P.: On the regularity and learnability of ordered DAG languages. In: Proc. 22nd International Conference on the Implementation and Application of Automata (CIAA'17). Lecture Notes in Computer Science, vol. 10329, pp. 27–39. Springer (2017)
4. Björklund, H., Drewes, F., Ericson, P.: Between a rock and a hard place – uniform parsing for hyperedge replacement DAG grammars. In: Dediu, A., Janoušek, J., Martín-Vide, C., Truthe, B. (eds.) Proc. 10th Intl. Conf. on Language and Automata Theory and Applications. Lecture Notes in Computer Science, vol. 9618, pp. 521–532 (2016)
5. Björklund, H., Drewes, F., Ericson, P., Starke, F.: Uniform parsing for hyperedge replacement grammars. Tech. Rep. UMINF 18.13, Umeå University, http://www8.cs.umu.se/research/uminf/index.cgi (2018), submitted for publication
6. Chiang, D., Andreas, J., Bauer, D., Hermann, K.M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: Proc. 51st Annual Meeting of the Association for Computational Linguistics (ACL 2013), Volume 1: Long Papers. pp. 924–932 (2013)
7. Courcelle, B.: The monadic second-order logic of graphs V: On closing the gap between definability and recognizability. Theoretical Computer Science **80**(2), 153–202 (1991)
8. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations, chap. 2, pp. 95–162. World Scientific (1997)
9. Drewes, F., Hoffmann, B., Minas, M.: Predictive top-down parsing for hyperedge replacement grammars. In: Proc. 8th Intl. Conf. on Graph Transformation (ICGT'15). Lecture Notes in Computer Science (2015)
10. Drewes, F., Hoffmann, B., Minas, M.: Predictive shift-reduce parsing for hyperedge replacement grammars. In: de Lara, J., Plump, D. (eds.) Proc. 10th Intl. Conf. on Graph Transformation (ICGT'17). Lecture Notes in Computer Science, vol. 10373, pp. 106–122 (2017)
11. Gilroy, S., Lopez, A., Maneth, S.: Parsing graphs with regular graph grammars. In: Proc. 6th Joint Conf. on Lexical and Computational Semantics (*SEM 2017). pp. 199–208 (2017)
12. Habel, A.: Hyperedge Replacement: Grammars and Languages, Lecture Notes in Computer Science, vol. 643. Springer (1992)
13. Habel, A., Kreowski, H.J.: May we introduce to you: Hyperedge replacement. In: Proceedings of the Third Intl. Workshop on Graph Grammars and Their Application to Computer Science. Lecture Notes in Computer Science, vol. 291, pp. 15–26. Springer (1987)