THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Probabilistic Graph Formalisms for Meaning Representations

Sorcha Gilroy

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2019

Abstract

In recent years, many datasets have become available that represent natural language semantics as graphs. To use these datasets in natural language processing (NLP), we require probabilistic models of graphs. Finite-state models have been very successful for NLP tasks on strings and trees because they are probabilistic and composable. Are there equivalent models for graphs? In this thesis, we survey several graph formalisms, focusing on whether they are probabilistic and composable, and we contribute several new results. In particular, we study the directed acyclic graph automata languages (DAGAL), the monadic second-order graph languages (MSOGL), and the hyperedge replacement languages (HRL). We prove that DAGAL cannot be made probabilistic, we explain why MSOGL also most likely cannot be made probabilistic, and we review the fact that HRL are not composable. We then review a subfamily of HRL and MSOGL: the regular graph languages (RGL; Courcelle 1991), which have not been widely studied, and particularly have not been studied in an NLP context. Although Courcelle (1991) only sketches a proof, we present a full, more NLP-accessible proof that RGL are a subfamily of MSOGL. We prove that RGL are probabilistic and composable, and we provide a novel Earley-style parsing algorithm for them that runs in time linear in the size of the input graph. We compare RGL to two other new formalisms: the restricted DAG languages (RDL; Björklund et al. 2016) and the tree-like languages (TLL; Matheja et al. 2015). We show that RGL and RDL are incomparable; TLL and RDL are incomparable; and either RGL are incomparable to TLL, or RGL are contained within TLL. This thesis provides a clearer picture of this field from an NLP perspective, and suggests new theoretical and empirical research directions.

Acknowledgements

I have so many people to thank for getting me to the point of writing this thesis. The incredible teachers, mentors, friends, and family who have been there to help me along the way really cannot know how grateful I am to them all.

Firstly, I'd like to thank my supervisors Adam Lopez, Sebastian Maneth, and Federico Fancellu. Adam has been an incredibly supportive and enthusiastic supervisor who I have had the pleasure and privilege of working with over the last four years. I can't imagine anyone else who would be willing to spend 5 hours straight talking about graphs. Sebastian was invaluable in helping me turn intuitive ideas into formal, theoretical proofs—something which was vital for this thesis. I have enjoyed working with Federico immensely, for teaching me how to actually apply these models and make them work, but also for chatting about Downton Abbey while we wait for results. I want to thank two collaborators in particular as well. Pijus Simonaitis, who was an intern here in 2015. He found the Courcelle paper that is cited (so many times) in this thesis, and really helped us to understand it. Ieva Vasiljeva, who was a masters student with Adam and I in 2017, she was great to work with and contributed heavily to the work appearing in the DAG automata chapter. I sincerely want to thank my examiners Shay Cohen and David Chiang for taking the time to read this thesis and for giving valuable feedback.

I am very lucky to be a part of several groups at Edinburgh. Firstly, the Centre for Doctoral Training in Data Science who took me in as a wide-eyed graduate and taught me so much. I want to thank EPSRC for providing me with funding for my masters and PhD. I want to thank the directors of the CDT: Charles Sutton, Chris Williams, Amos Storkey and Adam Lopez. I particularly want to thank the administrative staff Sally Galloway and Brittany Bovenzi. Being part of the CDT has certainly made the PhD experience less lonely, and I am so grateful my fellow students for making the last four years a lot of fun. Secondly, I feel incredibly lucky to be part of the EdinburghNLP group. Being part of such a dynamic and interesting group has been amazing, it's been great getting to know you all. Finally, I want to thank all of the members of Adam's AGORA group. This group has taught me how to be a researcher, how to communicate technical ideas with others, and has been so supportive every step of the way. I can't imagine how I would have finished the PhD without all of your help, especially all of the practice talks, paper reviews, and thesis chapters you have helped me with.

Outside of informatics, I have amazing friends who have supported me all the way through this process (and before). I want to thank my Cork friends Ciara, Sinéad,

Eithne, and Donagh for being there for the last 14 years, and always making me laugh. I want to thank the amazing friends I've made in Edinburgh through answering a gumtree ad for a flat four years ago. You have kept me sane during the difficult times and have organised the best themed parties that I don't think I will ever be able to recreate. Alex, Ally, Andi, Calum, Camille, Juliet, Matt, Rose, Scott, and Will, you have made the last four years so much fun.

I certainly would not have reached this point without the support of my amazing family. My parents instilled a thirst for knowledge in me from a young age, and are always there to encourage my next steps. I feel very lucky to have grown up in our family. Maeve, Séamus, Dara, and Cormac, I owe a great deal of this to you.

Finally, I have to thank Jack. You have been there since the beginning of this, and you have supported me every step of the way. I can't thank you enough for your encouragement during the difficult times. We have had so much fun and laughter over the last few years and I can't wait for many more.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

S. Gilroy 28/01/19

(*Sorcha Gilroy*)

Table of Contents

1 Introduction **1**
1.1 Meaning representations 1
1.2 Probabilistic graph formalisms 4
1.3 Contributions 8

2 Preliminaries **9**

3 Directed acyclic graph automata **13**
3.1 DAGs, DAG automata, and probability 14
3.1.1 DAG automata 15
3.1.2 Closure properties 20
3.1.3 Weighted and probabilistic DAG automata 21
3.2 Impossible probabilistic DAG automata 24
3.2.1 Multi-rooted DAGs 27
3.2.2 DAGs of unbounded degree 28
3.2.3 Planar DAG automata 28
3.3 Implications for semantic DAGs 31
3.4 Summary: DAG automata are not suitable for modelling graphs probabilistically 33
3.5 Open problems 33
3.5.1 Properties of DAG languages 33
3.5.2 Alternative weighting 36
3.6 Conclusions 38

4 Monadic second-order graph languages **41**
4.1 MSO logic on strings 42
4.2 MSO languages are closed under intersection 46
4.3 MSO on graphs 46

4.3.1 MS_1 . . . 47
4.3.2 MS_2 . . . 48
4.4 MSO transductions . . . 51
4.5 MSO and DAG automata . . . 58
4.6 Probabilistic MSO . . . 60
4.7 Conclusions . . . 60

5 Hyperedge replacement languages 63
5.1 Hyperedge replacement grammars . . . 63
5.2 HRGs are probabilistic but not intersectable . . . 68
5.3 Expressivity of HRGs . . . 69
5.4 HRLs are MSO transductions of RTLs . . . 70
5.5 Formal properties of HRGs . . . 74
5.6 Conclusions . . . 76

6 (Re)introducing regular graph languages 77
6.1 Informal description of RGG . . . 78
6.2 Formal definition of RGG . . . 79
6.3 Expressivity of RGG . . . 81
6.4 RGL is a subfamily of HRL and MSOGL . . . 85
6.4.1 Anchors and parameters . . . 88
6.4.2 Path properties of RGLs . . . 89
6.4.3 MSO for hypergraphs . . . 93
6.4.4 The precondition of the transducer . . . 94
6.4.5 RGLs satisfy the precondition . . . 102
6.5 Parsing as transduction . . . 103
6.5.1 The formulas of the transducer . . . 103
6.5.2 Transducer output and derivation trees . . . 105
6.6 RGLs are closed under intersection . . . 108
6.7 Conclusions . . . 110

7 Comparing RGG to other formalisms 111
7.1 Restricted DAG grammars . . . 111
7.2 Tree-like grammars . . . 113
7.3 RGL vs RDL vs TLL . . . 114
7.4 Conclusions . . . 116

8 Parsing regular graph grammars **119**
8.1 RGL recognition 122
8.1.1 Top-Down recognition for RGLs 122
8.2 Earley parsing 123
8.2.1 Connected ordering 131
8.2.2 Recognition complexity 132
8.3 Conclusions 133

9 Conclusions **135**
9.1 Open theoretical problems 136
9.2 Graph formalisms in practice 137

A Extra proofs from Chapter 6 **139**

Bibliography **145**

Chapter 1

Introduction

Natural language processing (NLP) involves teaching computers to understand, interpret, and generate human language. NLP systems for machine translation, summarisation, paraphrasing, and other tasks often fail to preserve the who-did-what-to-whom relationships, or compositional semantics, in sentences and documents because they model sentences as bags of words, or at best syntactic trees. In this thesis, we ask: *How can we define a probabilistic graph formalism to model meaning representations?*

1.1 Meaning representations

Take the German sentence "Anna fehlt ihrem Kater".[1] A word-for-word gloss of this sentence gives us the English "Anna is-missed-by her cat", and the more natural English translation is "Anna's cat misses her". However, when we passed this sentence through Google Translate, the output (until late 2017) was "Anna is missing her cat".[2] This mistake shows how machine translation systems do not always capture the compositional semantics of words in a sentence.

To correctly translate the sentence "Anna fehlt ihrem Kater" into English, we need to preserve four facts:

1. There is an individual named Anna.

2. There is a cat.

3. The cat is owned by Anna.

[1] Example from Jones et al. (2012).

[2] It is now "Anna is missing her hangover". Kater is a slang word for hangover in German.

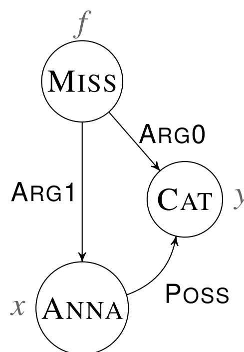
4. The cat misses Anna.

Google Translate gets facts 1-3 correct but gets fact 4 wrong since it says that Anna misses the cat. One way in which we could represent these facts is using first-order logic, which has been widely used to represent the meaning of sentences, as in the question answering system in Woods (1979).

$$\exists x.\exists y.\exists f.$$
$$\textsc{Anna}(x)\wedge \tag{1.1}$$
$$\textsc{Cat}(y)\wedge \tag{1.2}$$
$$\textsc{Poss}(x,y)\wedge \tag{1.3}$$
$$\textsc{Miss}(f)\wedge\textsc{Arg0}(f,y)\wedge\textsc{Arg1}(f,x) \tag{1.4}$$

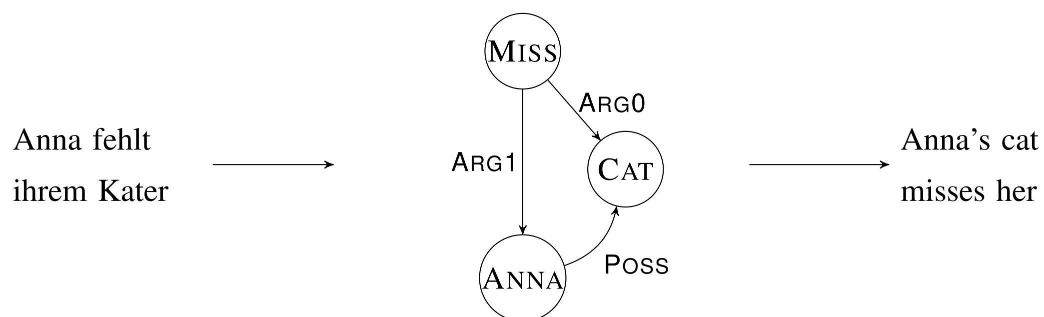
The equation numbers next to the logical statements correspond to the fact numbers listed above (i.e. Equation 1.2 corresponds to fact 2). For each character or event in the sentence, we have a variable: x is ANNA, y is CAT, and f is the missing event MISS. We then relate these variables to one another using the predicates POSS, ARG0, and ARG1. POSS represents possession, and ARG0 and ARG1 represent the semantic relationships *agent* and *patient* respectively.

We could also represent this logic in the form of a graph. The figure below shows a graph that covers these facts and can be defined from the logic. For each of the variables in the logic (x, y, and f), there is a corresponding node in the graph (the variables are not node labels but just indicators to show the connection to the logic). For each unary predicate over a variable v, node v is labelled by that predicate. For each binary predicate over variables v_1 and v_2, there is an edge from v_1 to v_2 labelled by the predicate name.



This graph is in the style of the Abstract Meaning Representation Bank (AMR; Banarescu et al. 2013), one of many **graph banks** that have been developed in recent years. Others include the Prague Czech-English dependency treebank (Hajič et al., 2012), Deepbank (Flickinger et al., 2012), the Universal Conceptual Cognitive Annotation (Abend and Rappoport, 2013), and the Groningen Meaning Bank (Bos, 2013). Surveys of these graphbanks can be found in Kuhlmann and Oepen (2016) and Abend and Rappoport (2017).[3] Since we have these graph banks, we would like to be able to use them in NLP applications such as machine translation or question answering.

Returning to our example, consider how we might use such a graph in machine translation (shown in the figure below). We first predict a semantic representation from the source sentence, and then predict a target sentence from this representation. To build such a semantic parser, we learn it from a dataset of sentences paired with their semantic representations. To do this learning, we need a probabilistic model. The large number of existing graph banks mean that we have the datasets. The natural question then is: *How can we define a probabilistic model that would allow us to map from strings to graphs (and vice-versa)?*



Since the release of the AMR graph bank, many systems have been created which map sentences to and from their AMR representations. Some of these systems convert the AMR annotation into a string and learn a string-to-string mapping (van Noord and Bos, 2017). Others adapt models used to predict dependency trees from sentences to predict AMRs from sentences (Wang et al., 2015; Damonte et al., 2017). The important point to note about these models is that by treating the AMR graphs as strings or trees, information in the graph banks is lost. The problem with this can be seen when their output is studied. Damonte et al. (2017) propose an evaluation for AMR that breaks an overall accuracy score down into smaller parts. They show that the highest accuracy

[3] A (growing) list of semantic banks organised by Emily Bender can be found here: http://bit.ly/2E2wruk.

achieved on **reentrancies** (i.e. nodes with multiple parents) was 42%. Since then, this number has increased but only to 52% (van Noord and Bos, 2017).[4]

One of the main motivations in creating the AMR graph bank was to have a representation which captures semantic relationships which are modelled by reentrancies, particularly those of control and coreference. Therefore, we want to define probabilistic models that explicitly model the objects in the AMR and other graph banks as graphs.

1.2 Probabilistic graph formalisms

Imagine we have a sentence $\mathbf{s}$ in a source language which we want to translate into some target language. The first step could be to use a string-to-graph model m_1 to predict the semantic graph $\mathbf{G}$ of $\mathbf{s}$ with probability $p(\mathbf{G} \mid \mathbf{s})$. If we wanted to then use a graph-to-string model m_2 to predict the target sentence $\mathbf{t}$, we would first need to check that $\mathbf{G}$ is in the domain of m_2, and if so then we could predict $\mathbf{t}$ with probability $p(\mathbf{t} \mid \mathbf{G})$. In general, there may be many candidate graphs $\mathbf{G}$, and so many possible translations of $\mathbf{s}$. This means that we would predict some set of graphs using m_1 and we would then intersect this set of graphs with the domain of m_2. Therefore, we would like to be able to compose m_1 and m_2, meaning that we should be able to compute the intersection of the output of the m_1 with the input domain of m_2.

Our challenge then is how to define models which (1) allow us to compute both $p(\mathbf{G} \mid \mathbf{s})$ and $p(\mathbf{t} \mid \mathbf{G})$, and (2) can be composed with one another. This leads then to the more fundamental question of how to define probability distributions over graphs, i.e. computing $p(\mathbf{G})$. If we can come up with such a family of models that generates graph languages that are closed under intersection, then we could define the conditional models $p(\mathbf{G} \mid \mathbf{s})$ and $p(\mathbf{s} \mid \mathbf{G})$ described in the translation pipeline.

If $\mathbf{G}$ were a string or tree instead of a graph, $p(\mathbf{G})$ could be computed using probabilistic finite automata (Mohri et al., 2008; Allauzen et al., 2014), which are closed under intersection. Finite automata have been used in NLP for applications such as speech recognition (Mohri et al., 2008), machine translation (Bangalore and Riccardi, 2001), and morphological analysis (Roark and Sproat, 2007).[5] This suggests that we

[4]Recently, Groschwitz et al. (2018) and Lyu and Titov (2018) model AMR as a graph. We briefly discuss these papers in Chapter 9.

[5]Finite-state models have recently been superseded by continuous relaxations in the form of neural network models. We will return to this point in the conclusion.

might want to look for the equivalent of finite automata on graphs, by answering the question: *How can we define a model that generates a probabilistic language of graphs that is also closed under intersection?* This question has not been widely studied in an NLP context, something we aim to do in this thesis.

There are many ways of defining models over graphs. In this thesis we will survey definitions based on automata, logic, and grammars. This is the first survey of its kind in an NLP context. While this thesis contributes to the understanding of this space, a complete understanding is beyond the scope of a single PhD thesis. While the thesis contains new results, we mention many open problems in this field, some with suggested approaches for tackling them. The models we deal with define languages of graphs mathematically using sets of rules for generation or constraints for recognition. For each of these models, we discuss the languages of graphs they generate or describe. A **language** L is a set of objects (e.g. strings, trees, or graphs). For example, a finite-state automaton A defines a regular string language by saying that L is the set of strings that are recognised by A. We will also discuss **families of languages**—collections of languages that can all be generated or recognised by the same formalism. For example, the regular string languages are a family of languages consisting of the set of all string languages L for which there exists some automaton A whose language is L.

Figure 1.1 shows a Hasse diagram describing the relationships between families of string, tree, and graph languages. We will now step through this diagram while describing the chapters of the thesis.

A lot of work has been carried out on families of string languages and their properties are well understood (see e.g. Hopcroft and Ullman 1979). Starting from the bottom right of Figure 1.1, we know that the regular string languages can be generated by finite-state automata, are closed under intersection, and have a probabilistic extension. They are a proper subfamily of the context-free string languages (shown on the bottom left of Figure 1.1), which have a probabilistic extension but are not closed under intersection. The story is very similar when we move up the diagram to trees. The regular tree languages can be generated by finite-state tree automata, they have a probabilistic extension, and they are closed under intersection. Again, they are a proper subfamily of the context-free tree languages which have a probabilistic extension but are not closed under intersection.

When we try to generalise these models to graphs, however, the story becomes a lot more complicated. A major problem is that it is not clear what a finite-state automaton or context-free grammar over graphs should look like. One possible extension of

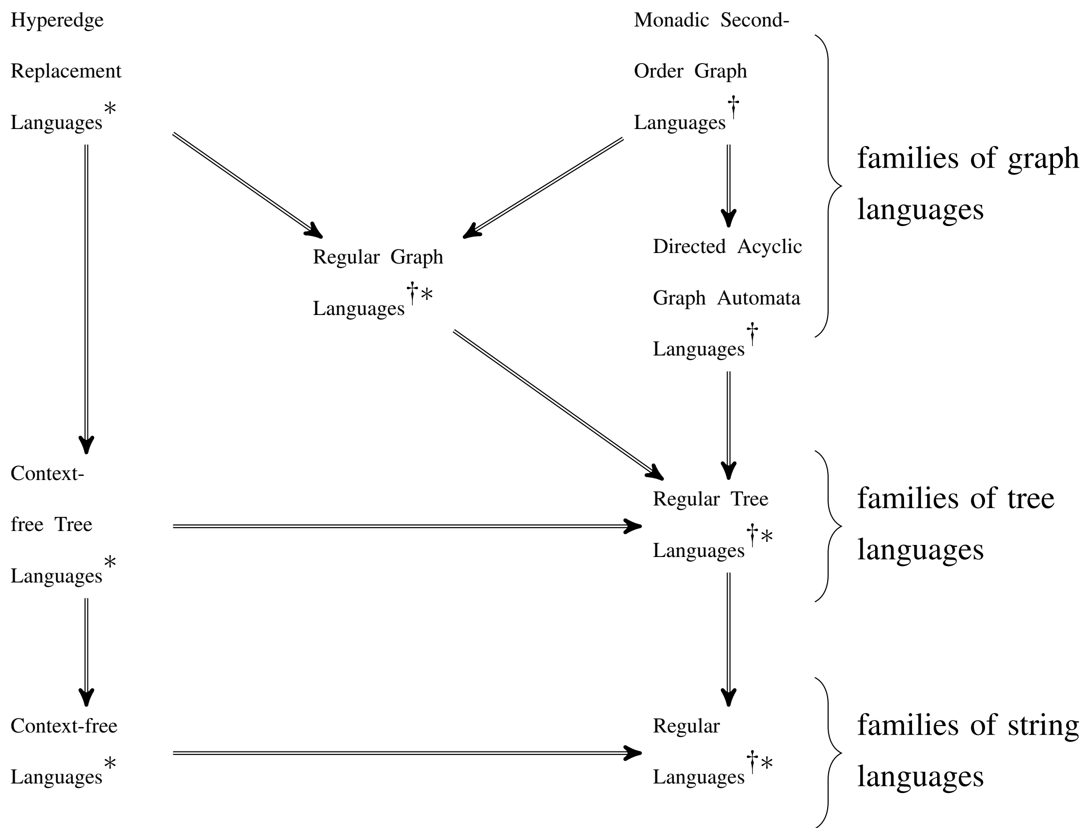


Figure 1.1: There is an arrow from A to B if B is a subfamily of A. † indicates languages which are closed under intersection, $*$ indicates languages which have probabilistic extensions.

FSA to graphs is the **directed acyclic graph automata** (DAG automata; Kamimura and Slutzki 1981), studied in an NLP context by and Quernheim and Knight (2012), Drewes (2017), and Chiang et al. (2018). This extension is based on the form of the transitions in FSAs, and the DAG automata languages are shown above the regular tree languages in Figure 1.1.[6] In Chapter 3, we provide a novel proof that the extension of *probabilistic* FSA to DAGs does not work in general (Vasiljeva et al., 2018). We exhibit a very simple DAG automaton that generates an infinite language of graphs, for which the only valid probability distribution we can define by weighting transitions is one in which the support is a single graph, with all other graphs receiving a probability of zero.

[6]The DAG automata languages of Kamimura and Slutzki (1981) and Quernheim and Knight (2012) are incomparable but both contain the regular tree languages and are subfamilies of the monadic second-order graph languages so we use one name for both families in this diagram. There is a detailed discussion of their relationship in Chapter 3.

Another possible extension of regular languages into graphs is by using **monadic second-order logic** (MSO; Courcelle and Engelfriet 2011). MSO is of special interest to us, since, when restricted to strings or trees, it exactly characterises the **recognisable** — or regular— languages of each (Büchi, 1960; Büchi and Elgot, 1958; Trakhtenbrot, 1966). MSO defines constraints which objects must satisfy but does not have an intuitive way of generating objects, unlike automata and grammars. It appears that the MSO graph languages may have no probabilistic extension since they contain the DAG automata languages (Thomas, 1991), as is shown in Figure 1.1. In Chapter 4, we will define MSO and discuss its properties, including why it is closed under intersection.

Moving up the left-hand side of the diagram, we look at how we can generalise context-free grammars to graphs. In particular, we discuss the **hyperedge replacement languages** (HRL; Drewes et al. 1997). HRL have already been studied in an NLP context by several researchers (Chiang et al., 2013; Peng et al., 2015; Bauer and Rambow, 2016). They share the properties of the other context-free languages, namely that they have a probabilistic extension but are not closed under intersection. This is in contrast to the MSO graph languages which are closed under intersection but may not be probabilistic. We will discuss HRL in Chapter 5, where we will also review why HRL and MSO graph languages are incomparable—a fact that distinguishes their relationship from the analogous relationship between context-free and regular string and tree languages.

At this point, it is natural to ask whether there is some subfamily of MSO graph languages and HRL that inherits the desirable properties of both. This leads us to reintroduce the **regular graph languages** (RGL; Courcelle 1991), whose definition is based on a restricted form of HRL and have not been studied in an NLP context, or much in general. In Chapter 6, we will define and discuss RGL and provide a detailed proof of why RGL languages are contained within the HRL and MSO graph languages (Gilroy et al., 2017b), which was only sketched in Courcelle (1991). We also give a novel proof that RGL is closed under intersection. This proof is quite general and may apply to many other formalisms.

There are other recent formalisms that appear to be related to RGL. They include the **restricted DAG languages** (RDL; Björklund et al. 2016) and the **tree-like languages** (TLL; Matheja et al. 2015). In Chapter 7, we compare the expressivity of RGL, RDL and TLL. As far as we know, this is the first comparison of any kind of these formalisms with one another.

After surveying each of these families of languages, we then will look at how

we might use these formal ideas in practice. In Chapter 8, we provide a novel top-down Earley style algorithm for recognising hyperedge replacement languages which is particularly efficient in recognising regular graph languages (Gilroy et al., 2017a). We prove that the parser is sound and complete.

1.3 Contributions

The contributions of this thesis are:

- We survey the landscape of families of graph languages, which is not nearly as well understood as it is for trees and strings. Most other discussions of this area lie mostly in the formal language theory literature, but our discussion is intended to be friendly to an NLP audience and organised around properties of interest to NLP practitioners.
- We exhibit a DAG automaton that generates an infinite language of graphs, for which the only valid probability distribution we can define by weighting transitions is one in which the support is a single graph, with all other graphs receiving a probability of zero. (This work is based on Vasiljeva et al., 2018).
- We provide a detailed proof that the regular graph languages are contained within both the hyperedge replacement languages and the monadic second-order graph languages, and we show that they inherit the desirable properties of both. (This work is based on Gilroy et al., 2017b).
- We compare the expressivity of the regular graph languages, the restricted DAG languages, and the tree-like languages.
- We generalise Earley's algorithm for parsing strings to hyperedge replacement languages which runs in linear time on regular graph languages. (This work is based on Gilroy et al., 2017a).

Chapter 2

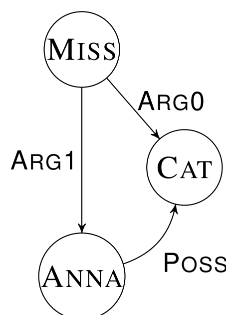
Preliminaries

We use this chapter to define some concepts and notation that will be used throughout the thesis. If A is a set then $s \in A^*$ denotes that s is a sequence of length at least 0, each element of which is in A. We denote by $|s|$ the length of s. If n is a positive number, then $[n]$ denotes the set $\{1,\ldots,n\}$.

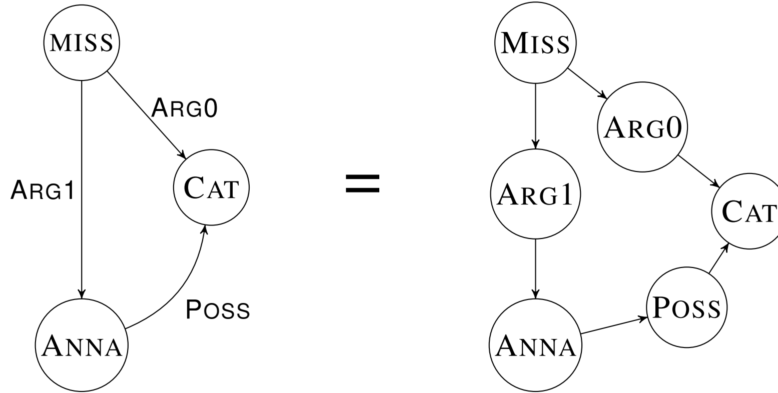
This thesis deals with graphs of slightly varying forms. We first define a directed graph with node and edge labels.

Definition 1. *A directed* ***graph*** *over node label set* Σ *and edge label set* Γ *is a tuple* $G = (V, E, lab_n, lab_e, src, tar)$ *where* V *is a finite set of nodes,* E *is a finite set of edges,* $lab_n : V \to \Sigma$ *is a function assigning labels to nodes,* $lab_e : V \to \Gamma$ *is a function assigning labels to edges,* src: $E \to V$ *is a function assigning a source node to every edge, and* tar: $E \to V$ *is a function assigning a target node to every edge.*

We are interested in graphs like the simplified AMR graph below, which represents entities and events as nodes, and relationships between them as edges. In this design, both nodes and edges have labels, and these labels represent the type of an entity, event, or relationship.



In Chapter 3, we deal with directed acyclic graph automata, which model graphs that only have labels on nodes. These node-labeled graphs can simulate edge labels using a node with one incoming and one outgoing edge, like so:



Sometimes we will discuss the set of edges coming into or going out of a node, so we define functions in: $V \to E^*$ and out: $V \to E^*$.

$$\text{in}(v) = \{e \mid \text{tar}(e) = v\}$$
$$\text{out}(v) = \{e \mid \text{src}(e) = v\}$$

The **degree** of a node is the number of edges connected to it, so the degree of v is $|\text{in}(v) \cup \text{out}(v)|$. A **path** in a directed graph from node v to node v' is a sequence of edges $(e_1, \dots, e_n)$ where $\text{src}(e_1) = v$, $\text{tar}(e_n) = v'$ and $\text{src}(e_{i+1}) = \text{tar}(e_i)$ for all i from 1 to $n-1$. A **cycle** in a directed graph is a path in which the first and last nodes are the same (i.e., $v = v'$). A directed graph without any cycles is a **directed acyclic graph (DAG)**.

A DAG is **connected** if every pair of its nodes is connected by an undirected path—a path which can traverse edges in either direction. A node with no incoming edges is called a **root**, and a node with no outgoing edges is called a **leaf**.

In Chapters 5 to 8, we deal with hypergraphs—graphs whose edges can connect any number of nodes. The hypergraphs we will use are edge-labelled but not node-labelled. They take their edge labels from a ranked alphabet, which is an alphabet A paired with a function rank: $A \to \mathbb{N}$

Definition 2. *A **hypergraph** over a ranked alphabet* Γ *is a tuple*

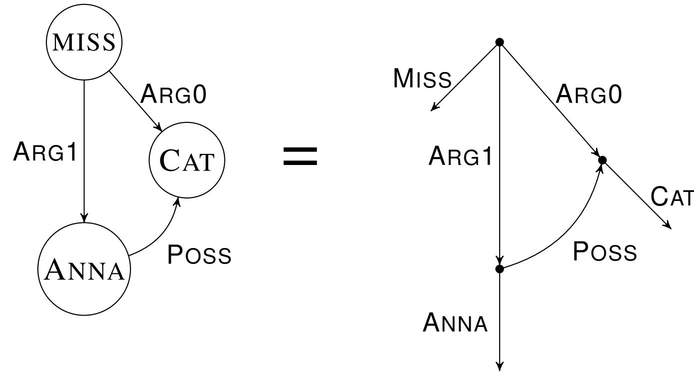
$$G = (V_G, E_G, att_G, lab_G, ext_G)$$

where V_G *is a finite set of nodes;* E_G *is a finite set of edges (distinct from* V_G*);* $att_G : E_G \to V_G^*$ *maps each edge to a sequence of nodes;* $lab_G : E_G \to \Gamma$ *maps each edge to a*

*label such that $|att_G(e)| = rank(lab_G(e))$; and ext_G is an ordered subset of V_G called the **external nodes** of G.*

We assume that the elements of ext_G are pairwise distinct. We also assume that for each edge e, the elements of $\text{att}_G(e)$ are pairwise distinct. An edge e is attached to its nodes by **tentacles**, each labelled by an integer indicating the node's position in $\text{att}_G(e) = (v_1, \ldots, v_k)$. The tentacle from e to v_i has label i, so the tentacle labels lie in the set $[k]$ where $k = \text{rank}(\text{lab}(e))$. To express that a node v is attached to the i-th tentacle of an edge e we say $\text{vert}(e, i) = v$. The nodes in ext_G are labelled by their position in ext_G. In figures, the i-th external node is labelled (i). We refer to nodes that are not external nodes as **internal nodes** and they are unmarked in figures. The **rank** of an edge e is k if $\text{att}(e) = (v_1, \ldots, v_k)$ (or equivalently, $\text{rank}(\text{lab}(e)) = k$). The **rank** of a hypergraph G is $|\text{ext}_G|$.

Hypergraphs will be used in hyperedge replacement grammars, which have terminal and nonterminal edges. In examples, we only show unary and binary terminal edges. We depict unary edges as directed edges with no target node, and binary edges as directed edges where the direction is determined by the tentacle labels: tentacle 1 attaches to the source and 2 attaches to the target. The hypergraphs we deal with have edge labels but not node labels. We simulate node labels in hypergraphs using unary edges as below:



We must redefine the notion of a path for use in hypergraphs. Given a hypergraph G, a **path** in G from a node v to a node v' is a sequence

$$(v_0, i_1, e_1, j_1, v_1)(v_1, i_2, e_2, j_2, v_2) \ldots (v_{k-1}, i_k, e_k, j_k, v_k)$$

such that $\text{vert}(e_r, i_r) = v_{r-1}$ and $\text{vert}(e_r, j_r) = v_r$ for each $r \in [k]$, $v_0 = v$, and $v_k = v'$. The length of this path is k. The degree of a node n in a hypergraph is the number of edges connected to that node, i.e. $\text{degree}(n) = |\{e | n \in \text{att}_G(e)\}|$.

Each of the formalisms we discuss in this thesis has an associated language that it generates or recognises. We use the notation that L refers to an abstract language and $\mathcal{L}$ is used as a function from a formalism to a language. For example, given an automaton A, $\mathcal{L}(A)$ is the language recognised by A.

Throughout this thesis, we will refer to weighted and probabilistic languages. A weighted language is a language L equipped with a function w such that $w(G)$ is the weight of each $G \in L$. We often wish this weight to define a probability distribution over the language.

A probability distribution over graphs is any function $p : L \to \mathbb{R}$ meeting two requirements:

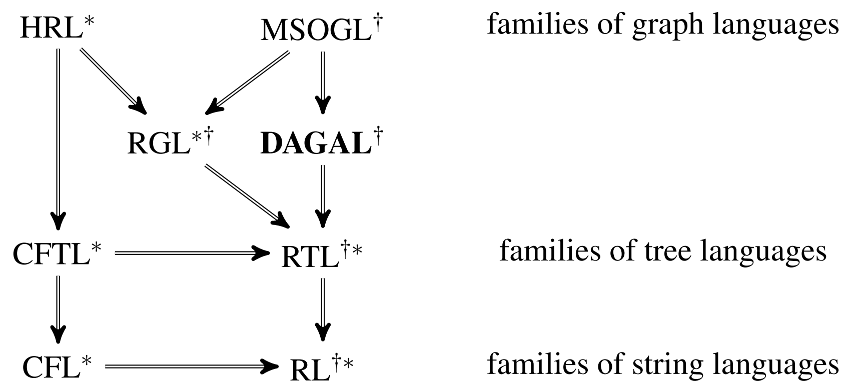
(R1) Every graph must have a probability between 0 and 1, inclusive. Formally, for all $G \in L$, $p(G) \in [0,1]$.

(R2) The probabilities of all graphs must sum to one. Formally, $\sum_{G \in L} p(G) = 1$.

This chapter serves the purpose of being a reference. Where relevant, we will repeat these definitions in the main chapters.

Chapter 3

Directed acyclic graph automata



We discussed in the introduction that we want to define an analogue of a probabilistic finite-state automaton (PFSA) for graphs. The vast majority of graphs that appear in semantic graph banks such as AMR are in the form of directed acyclic graphs (DAGs) and so we seek to define a PFSA for DAGs specifically.[1] One appealing contender is the DAG automaton (Kamimura and Slutzki, 1981), a natural extension of the finite tree automaton that generates planar DAGs—DAGs that can be drawn without any crossing edges.

Quernheim and Knight (2012) extended this DAG automaton with the explicit goal of modelling semantic graphs probabilistically, adding weights and removing the planarity constraint. Their automaton and its variants have been further studied with this goal firmly in mind (Blum and Drewes, 2016; Drewes, 2017; Chiang et al., 2018). But while Quernheim and Knight (2012) clearly intend for their weights to define probabil-

[1]The AMR bank does contain a small number of cyclic graphs but they allow inverted edges (e.g. converting an ARG0 edge in one direction to an ARG0-OF edge in the opposite directioni). By doing these inversions, every graph can be represented by a DAG, possibly with more than one root.

ities, they stop short of claiming that they do, instead ending their paper with an open problem: "Investigate a reasonable probabilistic model."

In this chapter, we investigate probabilistic models for DAG automata and prove a surprising result: *For some DAG automata, it is impossible to assign weights that define probability distributions with full support*. We exhibit a very simple DAG automaton that generates an infinite language of graphs, for which the only valid probability distribution we can define by weighting transitions is one in which the support is a single graph, with all other graphs receiving a probability of zero. This trivial distribution has little value in applications.

Our proof relies on the fact that a non-planar DAG automaton generates DAGs so prolifically that their number grows factorially in their size, rather than exponentially as in other automata. The proof holds under recent DAG automata variants that allow multiple roots or nodes of unbounded degree. But it breaks down when applied to the planar DAGs of Kamimura and Slutzki (1981), which are nevertheless too restrictive to model semantic graphs. Our result does not mean that it is impossible to define a probability distribution for the language that a DAG automaton generates, but it does mean that this distribution cannot be factored over the automaton's transitions, implying that convenient dynamic programming algorithms might not generalise to DAG automata that are powerful enough to model semantic graphs.

3.1 DAGs, DAG automata, and probability

Definition 1 in Chapter 2 has a formal description of DAGs. DAG automata operate over node-labelled DAGs defined as $G = (V, E, \text{lab}, \text{src}, \text{tar})$ where lab: $V \to \Sigma$ and Σ is an alphabet of node labels.

Because DAGs do not contain cycles, they must always have at least one root and one leaf, but they can have multiple roots and multiple leaves. The formal model in our main result in Theorem 1 requires DAGs to have only a single root, so, given a label set Σ, we distinguish between the set of all connected DAGs with a single root, $\mathcal{G}_\Sigma$; and those with one or more roots, $\mathcal{G}^*_\Sigma$.

Figure 3.1 shows the many varieties of DAG automata languages. The differences between the families depend on whether the DAGs in them are planar or non-planar, are single or multi-rooted, and have bounded or unbounded degree. Note that any automaton defines both a single- and a multi-rooted language. This is in contrast to the planar and non-planar languages which are generated by automata that are defined differently,

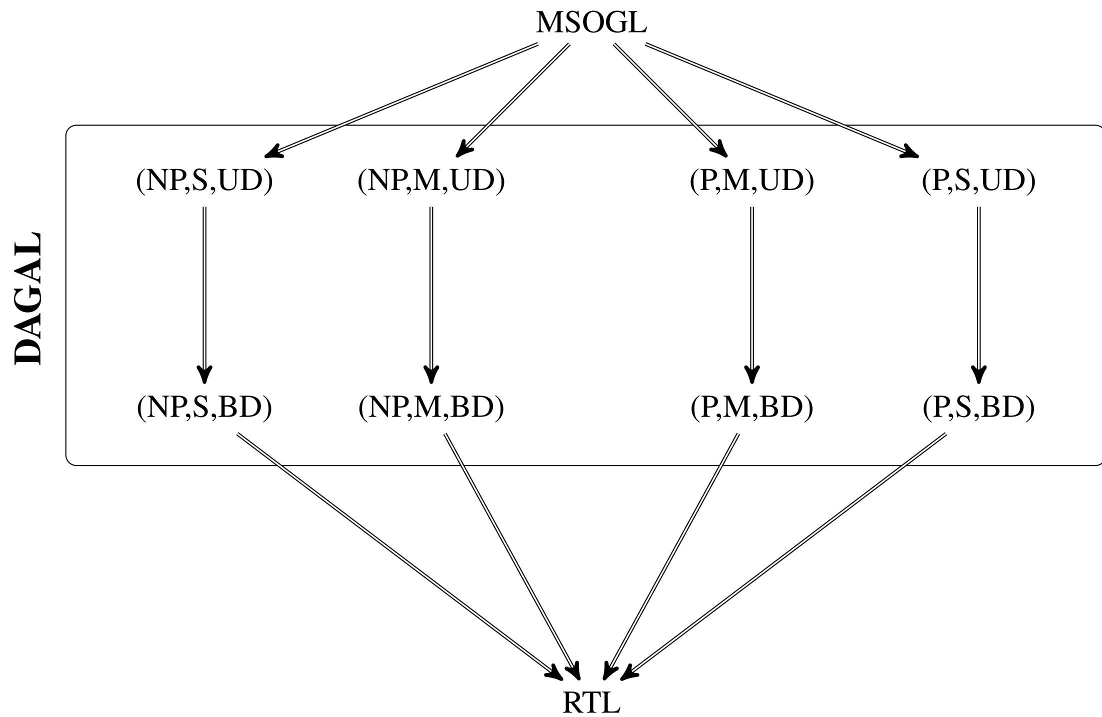


Figure 3.1: A detailed version of the right-hand side of Figure 1.1 showing how the different families of DAG automata languages (DAGAL) relate. Each family is described by a triple (a,b,c) where: a is P for planar or NP for non-planar; b is S for single or M for multiple; and c is UD for unbounded-degree or BD for bounded-degree.

and the same holds for the bounded and unbounded degree languages. Theorem 1, the central result of this chapter, applies to the single-rooted languages of non-planar DAG automata of bounded degree ((NP, S, BD) in Figure 3.1). We extend this result to the rest of the non-planar families shown on the left-hand side of the figure.

In §3.2.3, we discuss whether this result also applies to the planar DAG families, shown in the right-hand side of Figure 3.1. We show that this specific result does not apply and that the planar and non-planar families of DAG languages are incomparable. In §3.3 we discuss the implications of our results for semantic DAGs, and finally in §3.5 we consider some open problems in this area.

3.1.1 DAG automata

In finite automata over strings, symbols are generated sequentially by transitions from one state to another state. Tree finite automata generalise string finite automata by transitioning from one state to an ordered sequence of states if generating a tree top-

Automaton	Transitions	Example
string	one-to-one	$p \to p'$
top-down tree	one-to-many	$p \to (p', q')$
bottom-up tree	many-to-one	$(p', q') \to p$
planar DAG	many-to-many	$(p, q) \to (p', q')$
non-planar DAG	many-to-many	$\{p, q\} \to \{p', q'\}$

Table 3.1: The transitions for different varieties of automata. For example, one-to-many means that there can only be at most one state on the left-hand side but any number of states on the right-hand side. Parentheses indicate ordered tuples and curly-brackets indicate unordered multisets.

down from root to leaves, or from an ordered sequence of states to a single state if generating a tree bottom-up from leaves to root. The DAG automata of Kamimura and Slutzki (1981) generalise tree automata further, transitioning from one ordered sequence of states to another ordered sequence of states, and as we will see later, this produces planar DAGs (Section 3.2.3). Finally, the DAG automata of Quernheim and Knight (2012) transition from *multisets* of states to multisets of states, rather than from sequences to sequences, and this enables them to generate non-planar DAGs. These relationships are summarised in Table 3.1.

Until we formally define planar automata in §3.2.3, we will focus on non-planar DAG automata, and when we refer to DAG automata, we mean this type. To formally define them, we need some notation for multisets, which are sets that can contain repeated elements. A **multiset** is a pair (S, m) where S is a finite set and $m : S \to \mathbb{N}$ is a count function—that is, $m(x)$ counts the number of times x appears in the multiset. The set of all finite multisets over S is $M(S)$. When we write multisets, we will often simply enumerate their elements. For example, $\{p, q, q\}$ is the multiset containing one p and two qs, and since multisets are unordered, it is the same as both $\{q, p, q\}$ and $\{q, q, p\}$. We write $\emptyset$ for a multiset containing no elements.

Definition 3. *A **DAG automaton** is a triple $A = (Q, \Sigma, T)$ where Q is a finite set of states; Σ is a finite set of labels; and T is a finite set of transitions of the form $\alpha \xrightarrow{\sigma} \beta$ where $\sigma \in \Sigma$ is a node label, $\alpha \in M(Q)$ is the left-hand side, and $\beta \in M(Q)$ is the right-hand side.*

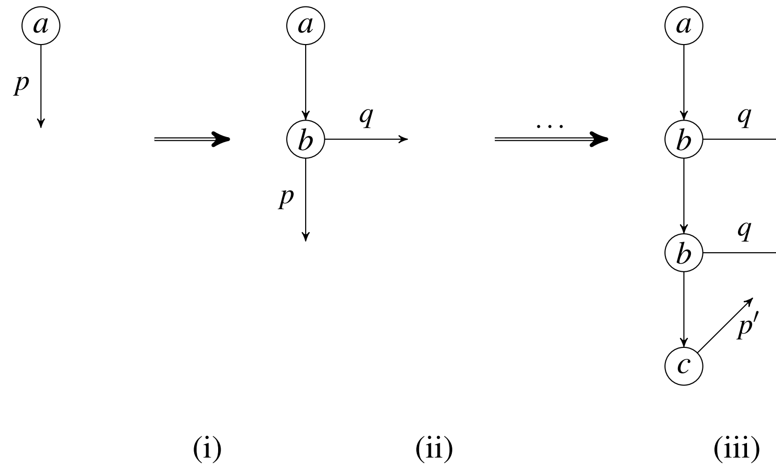


Figure 3.2: The first steps of a derivation of a DAG using the automaton A_{np} in Example 1. Double edges are applications of transitions, and those with ... over them indicate that more than one transition was applied. Edge labels in the DAGs indicate the frontier states. The root of the DAG is the node labelled a.

Example 1. Let $A_{np} = (Q, \Sigma, T)$ be a DAG automaton where $Q = \{p, p', q\}$, $\Sigma = \{a, b, c, d, e\}$, and the transitions in T are as follows:

$$\emptyset \xrightarrow{a} \{p\} \tag{t_1}$$

$$\{p\} \xrightarrow{b} \{p, q\} \tag{t_2}$$

$$\{p\} \xrightarrow{c} \{p'\} \tag{t_3}$$

$$\{p', q\} \xrightarrow{d} \{p'\} \tag{t_4}$$

$$\{p'\} \xrightarrow{e} \emptyset. \tag{t_5}$$

3.1.1.1 Generating single-rooted DAGs

A DAG automaton generates graphs from root to leaves. To illustrate this, we'll focus on the case where a DAG is allowed to have only a single root, and return to the multi-rooted case in Section 3.2.1. To generate the root, the DAG automaton can choose any transition with $\emptyset$ on its left-hand side—these transitions behave like transitions from the start state in a finite automaton on strings, and always generate roots. In our example, the only available transition is t_1, which generates a node labelled a with a dangling outgoing edge in state p, as in Figure 3.2(i). This dangling edge is the **frontier** of the partially-generated DAG.

While there are states on the frontier, the DAG automaton must choose a transition

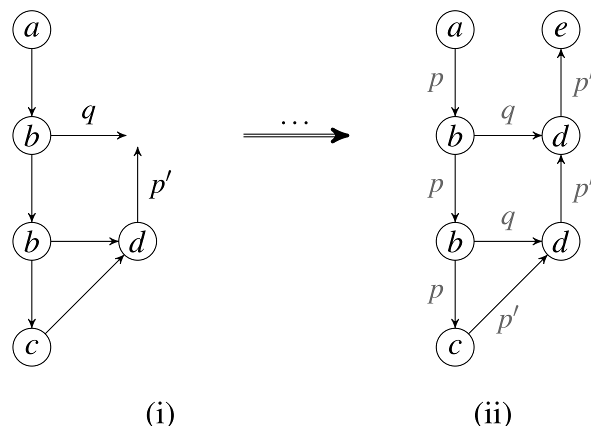


Figure 3.3: One continuation of the derivation shown in Figure 3.2 where p' first combines with the lowermost q and then the derivation is completed in a deterministic way.

whose left-hand side matches a subset of states on the frontier and apply it to generate a new node and a new set of frontier states. In our example, the automaton must choose between t_2 and t_3, and by choosing t_2, it arrives at the configuration in Figure 3.2(ii), with both a p and a q on the frontier and the incoming p state forgotten. Once again, it must choose between t_2 and t_3—it cannot use the q state because that state can only be used by t_4, which also requires a p' on the frontier. So each time it applies t_2, the choice between t_2 and t_3 repeats.

If the automaton chooses t_3, as it has done in Figure 3.2(iii), it has a new set of choices, between t_4 and t_5. But notice that choosing t_5 will leave the q states stranded, leaving a partially derived DAG; we consider a run of the automaton successful only when the frontier is empty, so this choice leads to a dead-end path.

If the automaton chooses t_4, it has an additional choice: it can combine p' with *either* of the available q states. If it combines with the lowermost q, it can then apply t_4 to consume the remaining q, followed by t_5, which has $\emptyset$ on its right-hand side (Figure 3.3). If instead the p' state first combines with the upper q, a different DAG is generated (Figure 3.4). Transitions with $\emptyset$ on their right-hand sides behave like transitions to a final state in a finite automaton, and generate leaf nodes.

While the DAGs in Figure 3.3(ii) and Figure 3.4(ii) are planar, this DAG automaton can produce non-planar DAGs, as in Figure 3.5. This graph has a minor graph—a graph that results by contracting some edges—called $K_{3,3}$, the complete (undirected) bipartite graph over two sets of three nodes, and any graph with a $K_{3,3}$ minor is non-planar.

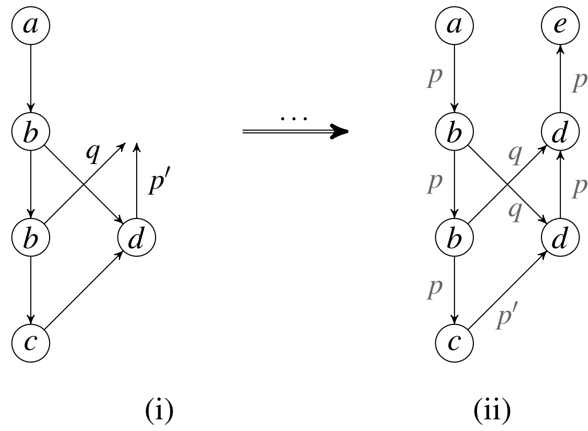


Figure 3.4: An alternative continuation of the derivation shown in Figure 3.2 where p' first combines with the uppermost q.

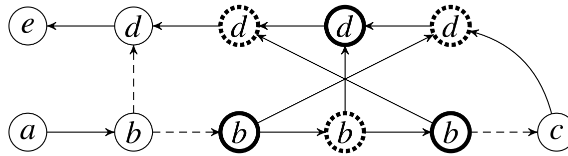


Figure 3.5: A non-planar DAG generated by the automaton A_{np} in Example 1. The bold and dotted nodes are two sets of nodes that form the minor $K_{3,3}$ when the dashed edges are contracted.

3.1.1.2 Recognising DAGs and DAG languages

We can precisely define the language generated by a DAG automaton in terms of recognition, which asks whether an input DAG could have been generated by an input automaton. We recognise a DAG by finding a run of the automaton that could have generated it. We can guess a run on a DAG by guessing a state for each of its edges, and then ask whether those states simulate a valid sequence of transitions.

A **run** of DAG automaton $A = (Q, \Sigma, T)$ on DAG $G = (V, E, \text{lab}, \text{src}, \text{tar})$ is a mapping $\rho : E \to Q$ from edges of G to automaton states Q. A run is **accepting** if for all $v \in V$ there is a corresponding transition $\rho(\text{in}(v)) \xrightarrow{\text{lab}(v)} \rho(\text{out}(v))$ in T, we extend ρ to multisets where $\rho(\{e_1, \ldots, e_n\}) = \{\rho(e_1), \ldots, \rho(e_n)\}$. DAG G is **recognised** by automaton A if there is an accepting run of A on G.

Example 2. The DAGs in Figure 3.3(ii) and 3.4(ii) can be recognised by the automaton A_{np} in Example 1. The only accepting run is written as grey edge labels on the graph.

Given a DAG automaton A, we define its single-rooted language $\mathcal{L}_s(A)$ as

$$\mathcal{L}_s(A) = \{G \in \mathcal{G}_\Sigma \mid A \text{ accepts } G\}.$$

3.1.2 Closure properties

We mentioned in the introduction that we are interested in models of graphs that are closed under intersection. DAG automata are closed under both union and intersection. We will give these constructions here. Let $A_1 = (Q_1, \Sigma_1, T_1)$ and $A_2 = (Q_2, \Sigma_2, T_2)$ be DAG automata.

Then $A_\cup = (Q_\cup, \Sigma_\cup, T_\cup)$ is the union of A_1 and A_2 and is defined as follows:

$$Q_\cup = Q_1 \cup Q_2,$$

$$\Sigma_\cup = \Sigma_1 \cup \Sigma_2,$$

$$T_\cup = T_1 \cup T_2.$$

We assume here that Q_1 and Q_2 are disjoint, if not then we can rename each $q \in Q_2$ to q' to distinguish from $q \in Q_1$. For any DAG $G \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, an accepting run of G under A_1 or A_2 will also be an accepting run under $A_\cup$. Similarly, for any DAG $G \in \mathcal{L}(A_\cup)$, an accepting run of G under $A_\cup$ will be an accepting run under at least one of A_1 or A_2.

The construction for intersection is essentially a cross-product of the two input automata. It is slightly more complicated than the union construction and is defined as follows, $A_\cap = (Q_\cap, \Sigma_\cap, T_\cap)$. The states of $A_\cap$ are defined as

$$Q_\cap = \{q_1 q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}.$$

The alphabet is

$$\Sigma_\cap = \Sigma_1 \cap \Sigma_2.$$

And for each pair of transitions

$$\{p_1, \ldots, p_n\} \xrightarrow{\sigma} \{q_1, \ldots, q_m\} \in T_1 \text{ and } \{p'_1, \ldots, p'_n\} \xrightarrow{\sigma} \{q'_1, \ldots, q'_m\} \in T_2,$$

we construct transitions

$$\{\{p_{i_1} p'_{j_1}, \ldots, p_{i_n} p'_{j_n}\} \xrightarrow{\sigma} \{q_{k_1} q'_{l_1}, \ldots, q_{k_m} q'_{l_m}\} \mid$$
$$\{i_1, \ldots, i_n\} = \{j_1, \ldots, j_n\} = [n] \text{ and } \{k_1, \ldots, k_m\} = \{l_1, \ldots, l_m\} = [m]\}.$$

These transitions make up $T_\cap$. The reason we have a set of transitions in $T_\cap$ for a pair of transitions from A_1 and A_2 is because the order of the states doesn't matter, and so we need to pair up each state in $\{p_1,\dots,p_n\}$ with each state in $\{p'_1,\dots,p'_n\}$.

Example 3. Say we have

$$\{p_1,p_2\} \xrightarrow{\sigma} \{q_1,q_2\} \in T_1 \text{ and } \{p'_1,p'_2\} \xrightarrow{\sigma} \{q'_1,q'_2\} \in T_2,$$

then in $T_\cap$ we would have the four transitions:

$$\{p_1p'_1,p_2p'_2\} \xrightarrow{\sigma} \{q_1q'_1,q_2q'_2\},\{p_1p'_1,p_2p'_2\} \xrightarrow{\sigma} \{q_1q'_2,q_2q'_1\},$$
$$\{p_1p'_2,p_2p'_1\} \xrightarrow{\sigma} \{q_1q'_1,q_2q'_2\},\{p_1p'_2,p_2p'_1\} \xrightarrow{\sigma} \{q_1q'_2,q_2q'_1\}.$$

Let G be a DAG in $\mathcal{L}(A_1)\cap\mathcal{L}(A_2)$ with accepting runs ρ_1 under A_1 and ρ_2 under A_2. For each $e \in E_G$ define $\rho_\cap(e) = \rho_1(e)\rho_2(e)$. Then $\rho_\cap$ is an accepting run for G under $A_\cap$. Similarly, let $G \in \mathcal{L}(A_\cap)$, then there is an accepting run $\rho_\cap$ recognising G under $A_\cap$. For each $p \in Q_\cap$, $p = p_ip'_j$, and so for each $e \in E_G$, if $\rho_\cap(e) = p_ip'_j$ define $\rho_1(e) = p_i$ and $\rho_2(e) = p'_j$. Then ρ_1 and ρ_2 are both accepting runs of G under A_1 and A_2, respectively.

3.1.3 Weighted and probabilistic DAG automata

Now that we have shown that DAG automata languages are closed under intersection, we would like to define a probability distribution over a graph language L. Recall from Chapter 2, a probability distribution over graphs is any function $p : L \to \mathbb{R}$ meeting two requirements:

(R1) Every graph must have a probability between 0 and 1, inclusive. Formally, for all $G \in L$, $p(G) \in [0,1]$.

(R2) The probabilities of all graphs must sum to one. Formally, $\sum_{G\in L} p(G) = 1$.

R1 and R2 suffice to define a probability distribution, but in most learning applications we are interested in a slightly stronger statement of R1: we would like for all graphs to receive a *non-zero* weight, since in practical applications, an object with probability zero is effectively not in the language. We say that p has the **full support** of L if and only if it meets this stronger condition.

(R1') Every graph must have a probability greater than 0 and less than or equal to 1. Formally, for all $G \in L$, $p(G) \in (0,1]$.

While there are many ways to define a function that meets requirements R1' and R2, probability distributions in natural language processing are nearly universally defined in terms of weighted automata or grammars, so we adapt a common definition of weighted grammars (Booth and Thompson, 1973) to DAG automata.

We assign weights to DAGs using semirings. A semiring is a set $\mathbb{K}$ equipped with two binary operations $\oplus$ and $\otimes$. $\oplus$ is commutative with identity element $\bar{0}$, $\otimes$ has identity element $\bar{1}$, $\otimes$ distributes over $\oplus$, and $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$. We write a semiring as the tuple $(\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})$. A commonly used semiring, which we use here, is the semiring over the reals where $\oplus$ is normal addition, $\otimes$ is multiplication, $\bar{0} = 0$, and $\bar{1} = 1$.

Definition 4. *A **weighted DAG automaton** over a semiring $\mathbb{K}$ is a pair (A, w) where $A = (Q, \Sigma, T)$ is a DAG automaton and $w : T \to \mathbb{K}$ is a function that assigns weights in $\mathbb{K}$ to the transitions of A.*

From now on in this chapter, we will use the real semiring, $\mathbb{R}$. Since weights are functions of transitions, we will write them on transitions following the node label and a slash (/). For example, if $p \xrightarrow{a} q$ is a transition and 2 is its weight, we write $p \xrightarrow{a/2} q$.

Example 4. Let (A, w) be a weighted DAG automaton with $A = (Q, \Sigma, T)$, where $Q = \{p, q\}$, $\Sigma = \{a, b, c\}$, and the weighted transitions of T are as follows:

$$\emptyset \xrightarrow{a/0.5} \{p, q\} \qquad (t'_1)$$

$$\{p\} \xrightarrow{b/0.5} \{p\} \qquad (t'_2)$$

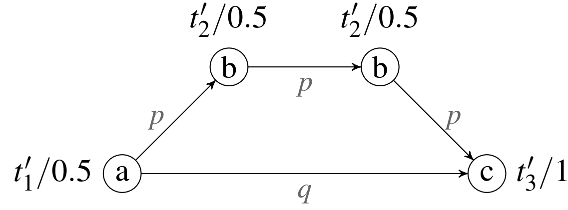
$$\{p, q\} \xrightarrow{c/1} \emptyset \qquad (t'_3)$$

We use the weights on transitions to weight runs.

Definition 5. *Given a DAG $G = (V, E, lab, src, tar)$, and an accepting run ρ of a weighted DAG automaton (A, w), we extend w to compute the **weight of the run** $w(\rho)$ by multiplying the weights of all of its transitions:*

$$w(\rho) = \prod_{v \in V} w(\rho(in(v)) \xrightarrow{lab(v)} \rho(out(v))).$$

Example 5. Given the DAG automaton of Example 4 and the DAG in the figure below, the weight of its only accepting run is $0.5 \times 0.5 \times 0.5 \times 1 = 0.125$.



Let $R_A(G)$ be the set of accepting runs of a DAG G using the automaton A. We extend w again to calculate the weight of a DAG, G, as the sum of the weights of all the runs in $R_A(G)$:

$$w(G) = \sum_{\rho \in R_A(G)} w(\rho).$$

While all weighted DAG automata assign real values to DAGs, not all weighted DAG automata define probability distributions. To do so, they must also satisfy requirements R1 and R2.

Definition 6. *A weighted automaton (A, w) is* ***probabilistic*** *if and only if the extension of w to $\mathcal{L}(A)$ is probabilistic.*

Example 6. Consider the weighted automaton in Example 4. Every DAG generated by this automaton must use t_1' and t_3' exactly once, and can use t_2' any number of times. Let G_n be the DAG that uses t_2' exactly n times. Then the language L defined by this automaton is $\bigcup_{n\in\mathbb{N}} G_n$ and since $w(G_n) = w(t_1')w(t_2')^n w(t_3')$, we have:

$$\begin{aligned} w(\bigcup_{n\in\mathbb{N}} G_n) &= \sum_{n=0}^{\infty} w(G_n) \\ &= \sum_{n=0}^{\infty} w(t_1')w(t_2')^n w(t_3') \\ &= \sum_{n=0}^{\infty} 0.5^{n+1} = 1. \end{aligned}$$

Hence the weighted DAG automaton in Example 4 is also a probabilistic DAG automaton.

Definition 7. *A weighted automaton (A, w) is* ***probabilistic with full support*** *if and only if the extension of w to $\mathcal{L}(A)$ is probabilistic with full support.*

For every finite automaton over strings or trees, there is a weighting of its transitions that makes it probabilistic (Booth and Thompson, 1973), and it is easy to show

that there is also a weighting that makes it probabilistic with full support. For example, if for every state in a weighted finite automaton on strings, the sum of weights on its outgoing transitions is 1 and each weight is greater than 0, then it is probabilistic with full support.[2] But as we will show, for some DAG automata there are no weights that make them probabilistic with full support.

3.2 Impossible probabilistic DAG automata

We will construct a DAG automaton that can generate a factorial number of graphs for a given number of nodes, and we will show that for any non-trivial assignment of weights, this factorial growth rate causes the weight of all graphs to sum to infinity.

Let $A_{np} = (Q, \Sigma, T)$ be the automaton defined in Example 1 with weight function $w : T \to \mathbb{R}$.

Theorem 1. *There is no assignment of weights to the DAG automaton A_{np} that defines a probability distribution with full support over its single-rooted language $\mathcal{L}_s(A_{np})$.*[3]

Proof. In any run of the automaton, transition t_1 is applied exactly once to generate the single root, producing a p on the frontier. The run can then choose between t_2 and t_3. If it chooses t_2, it keeps a single p on the frontier and adds a q, and must then repeat the same choice. Suppose that the automaton chooses t_2 exactly n times in succession, and then chooses t_3. At this point, the frontier contains n edges in state q and one in state p'.

The only way to consume all the frontier states is to choose transition t_4 exactly n times, consuming a q at each step, followed by t_5 to consume p' and complete the derivation. Therefore, for any graph in $\mathcal{L}_s(A_{\text{np}})$, t_1, t_3 and t_5 are each applied once, and t_2 and t_4 are each applied n times.

In general, when the automaton applies t_4 for the first time, it has n choices of q state to consume, each distinguished by its unique path from the root. The second application of t_4 has $n-1$ choices of q, and the ith application of t_4 has $n-(i-1)$ choices. Therefore, there are $n!$ ways to consume the q states, each producing a unique graph.

[2]Assuming no epsilon transitions, in our notation for DAG automata restricted to the string case this would include transitions to $\emptyset$, which correspond to final states with a final probability of 1 (Mohri et al., 2008).

[3]This proof uses the arithmetic semiring over the reals. We looked into whether there is some other semiring that could generate a probability distribution but could not find any that work.

Let $f(n)$ be the weight of a run where t_2 has been applied n times, and let $c(n)$ be the number of unique runs where t_2 has been applied n times.

$$\begin{aligned} f(n) &= w(t_1)w(t_2)^n w(t_3)w(t_4)^n w(t_5) \\ c(n) &= n! \end{aligned}$$

Now we claim that any DAG in $\mathcal{L}_s(A_{\text{np}})$ has exactly one accepting run, because every node label maps to a unique transition, so every pair of nodes connected by an edge determines a unique state in the accepting run. For example, an edge from a b node to a c node must be labelled p in any accepting run. So we have:

$$w(G) = f(n).$$

Let L_n be the set of graphs recognised by A_{np} where t_2 has been applied n times. Then $\mathcal{L}_s(A_{\text{np}}) = \bigcup_{n=0}^{\infty} L_n$ and $w(L_n) = f(n)c(n)$. Therefore,

$$\begin{aligned} w(\mathcal{L}_s(A_{\text{np}})) &= \sum_{G \in \mathcal{L}_s(A_{\text{np}})} w(G) \\ &= \sum_{n=0}^{\infty} w(L_n) \\ &= \sum_{n=0}^{\infty} f(n)c(n) \\ &= \sum_{n=0}^{\infty} \left[w(t_1)w(t_2)^n w(t_3)w(t_4)^n w(t_5) \right] [n!]. \end{aligned}$$

Let $B = w(t_1)w(t_3)w(t_5)$, and $C = w(t_2)w(t_4)$. Then,

$$w(\mathcal{L}_s(A_{\text{np}})) = \sum_{n=0}^{\infty} BC^n n!.$$

To make A_{np} a probabilistic DAG automaton with full support, R1' and R2 require us to choose B and C so that $w(\mathcal{L}(A_{\text{np}})) = 1$ and $BC^n \in (0,1]$ for all n. Note that we don't require the component weights of B or C to be in this interval—they can be any real numbers. However, since we require BC^n to be positive for all n, both B and C must be positive.

Now we will show that any choice of positive B and positive C satisfying R1' and R2 causes $w(\mathcal{L}_s(A_{\text{np}}))$ to diverge, using the ratio test (D'Alembert, 1768).

Let S be an infinite series of the form $\sum_{n=0}^{\infty} a_n$, and let $\alpha = \lim_{n\to\infty} \frac{|a_{n+1}|}{|a_n|}$. The **ratio test** says that S converges if $\alpha < 1$ and S diverges if $\alpha > 1$. It is inconclusive if $\alpha = 1$. In our case:

$$\lim_{n\to\infty} \frac{|BC^{n+1}(n+1)!|}{|BC^n n!|} = \lim_{n\to\infty} |C|(n+1) = \infty$$

since $C \neq 0$.

Therefore, $w(L) = \infty$ for any choice of positive B and positive C. So there is no assignment of weights to the transitions of A_{np} that defines a probability distribution over $\mathcal{L}_s(A_{\text{np}})$ with full support. □

Note that any automaton recognising $\mathcal{L}_s(A_{\text{np}})$ must accept factorially many DAGs in the number of nodes. Our proof implies that there is *no* probabilistic DAG automaton for language $\mathcal{L}_s(A_{\text{np}})$, since no matter how we design its transitions—which must be isomorphic to those in A_{np} apart from the identities of the states—the factorial will eventually overwhelm the constant factor corresponding to C in our proof, no matter how small we make it.

Theorem 1 does not mean that it is impossible to define a probability distribution over $\mathcal{L}_s(A_{\text{np}})$. It requires condition R1'—if we require only condition R1, then a solution of B=1 and C=0 makes the automaton probabilistic. However, this trivial distribution is not particularly useful: it assigns all of its mass to the singleton language $\{ⓐ \rightarrow ⓒ \rightarrow ⓔ\}$.

Theorem 1 also does not mean that it is impossible to define a probability distribution over $\mathcal{L}_s(A_{\text{np}})$ with full support. Suppose that for every graph G with n t_2 transitions, we let $p(G) = \frac{1}{2^{n+1}n!}$. Then,

$$w(\mathcal{L}_s(A_{\text{np}})) = \sum_{n=0}^{\infty} \frac{1}{2^n n!} n! = \sum_{n=0}^{\infty} \frac{1}{2^n} = 1.$$

However, this distribution does not factor over the transitions of any automaton, so we cannot generalise classic dynamic programming algorithms for probabilistic inference with this distribution.

In practical settings where the generation of a DAG is conditioned on some other object, for example a sentence in the case of semantic parsing, it is common to define a conditional probability distribution over a finite set of graphs. In this case we can define a probability distribution over these graphs since each has finite weight and there are finitely many of them. We can simply normalise the weights by their sum to define a probability distribution conditioned on some input. What we show here is that there is no way to globally normalise the distribution over all possible graphs since the sum of their weights is infinite.

3.2.1 Multi-rooted DAGs

What happens when we consider DAG languages that allow multiple roots? In one reasonable interpretation of AMRbank, more than three quarters of the DAGs have multiple roots (Kuhlmann and Oepen, 2016), so we want a model that permits this.[4]

Recall that for automaton A we defined a single-rooted language $\mathcal{L}_s(A) = \{G \in \mathcal{G}_\Sigma \mid A \text{ accepts } G\}$. We now define the multi-rooted language $\mathcal{L}_m(A)$:

$$\mathcal{L}_m(A) = \{G \in \mathcal{G}_\Sigma^* \mid A \text{ accepts } G\}.$$

Section 3.1.1.1 defined the generation of a single-rooted DAG using an automaton. To generate DAGs with multiple roots, we simply allow the application of a start transition (i.e. one with $\emptyset$ on its left-hand side) to be applied at any time. We still require the resulting DAGs to be connected. We first deal with multi-rooted DAG languages of bounded degree ((NP,M,BD) in Figure 3.1).

Although single- and multi-rooted DAG languages can be defined by a single automaton, they differ in an important way: they have different path languages. The **path language** of a DAG is the set of strings that label any path from a root to a leaf, and the path language of a DAG automaton includes the path language of every DAG it generates. For example, the path language of the graph in Figure 3.3 is $\{abde, abbdde, abbcdde\}$. Berglund et al. (2017) show that multi-rooted DAG automata have regular path languages, while single-rooted DAG automata have much more expressive path languages, which they argue are too powerful for modelling semantics.[5]

More interestingly, Drewes (2017) makes the same observation about path languages in single-rooted DAG languages using a construction very similar to the one in Theorem 1, so it's natural to wonder whether the problem with path languages and the problem with probabilities in single-rooted DAG languages have the same underlying cause. We now show that they do not, because any multi-rooted language contains the single-rooted language as a sublanguage. We make this notion precise below.

Corollary 1. *There is no assignment of weights to the DAG automaton A_{np} that defines a probability distribution over its multi-rooted language $\mathcal{L}_m(A_{np})$.*

[4] Although AMR annotations are single-rooted, they achieve this using a duplicated edge label set: the first set contains labels like ARG0; while the second contains their inverse, like ARG0-OF. The number cited here assumes edges of the second type are converted to the first type by reversing their direction.

[5] The single-rooted path languages are characterised by a partially blind multi-counter automaton.

Proof. By their definitions, $L_s(\mathcal{A}) \subset L_m(\mathcal{A})$, so:

$$\sum_{G \in L_m(\mathcal{A})} w(G) = \sum_{G \in L_s(\mathcal{A})} w(G) + \sum_{G \in L_m(\mathcal{A}) \setminus L_s(\mathcal{A})} w(G)$$

The first term is ∞ by Theorem 1 and the second is positive by R1', so the sum diverges, and there is no w for which $(\mathcal{A}, w)$ is probabilistic with full support over $L_m(\mathcal{A})$. □

It should be noted that although for a given automaton, its single-rooted language is a sublanguage of its multi-rooted language, the families of single and multi-rooted DAG automata languages are incomparable. Since the single-rooted DAG languages only contain languages of DAGs with exactly one root, there are clearly languages within the multi-rooted DAG languages that are not in the family of single-rooted DAG languages. In the opposite direction, the multi-rooted languages cannot be restricted to just a single root. The languages will contain all possible numbers of roots allowed, and so the multi-rooted DAG languages do not contain all the single-rooted DAG languages.

3.2.2 DAGs of unbounded degree

The maximum degree of any node in any DAG recognised by a DAG automaton is bounded by the maximum number of states in any transition, because any transition $\alpha \xrightarrow{\sigma} \beta$ generates a node with $|\alpha|$ incoming edges and $|\beta|$ outgoing edges. So, the families of DAG languages we have considered all have bounded degree.

DAG languages with unbounded degree could be useful to model phenomena like coreference in meaning representations, and they have been studied by Quernheim and Knight (2012) and Chiang et al. (2018). Consider both the single- and multi-rooted DAG languages generated by non-planar DAG automata of unbounded degree. These families are shown as (NP,S,UD) and (NP,M,UD) in Figure 3.1 and contain the families of single- and multi-rooted DAG languages generated by DAG automata of bounded degree, respectively. Therefore, by Theorem 1, these families contain DAG automata that cannot be made probabilistic. As in Corollary 1, this shows that the DAG languages of unbounded degree also cannot always be made probabilistic.

3.2.3 Planar DAG automata

We now turn to the right-hand side of Figure 3.1, the planar DAG automata. They were introduced in Kamimura and Slutzki (1981), before Quernheim and Knight (2012) ex-

tended them to non-planar automata for use in NLP. The question then is whether the problem shown in Theorem 1 applies to planar automata. The fundamental problem with trying to assign probabilities to DAG automata is the factorial growth in the number of DAGs with respect to the number of nodes. Does this problem occur in planar DAGs?

Planar DAG automata are similar to the DAG automata defined in Section 3.1 but with an important difference: rather than transition between multisets of states, they transition between *ordered* sequences of states. We write these sequences with parentheses, as in (p,q), which is distinct from (q,p), and we write ε for empty sequences. Planar DAG automata recognise only ordered DAGs, unlike the unordered DAGs we have seen so far. In generating DAGs using planar automata, we always have a strict order over the set of frontier states. If we want to apply a transition with (p,q) on the left-hand side, then we could only combine two edges in states p and q if the p edge immediately precedes the q edge in the order. Each transition replaces a subsequence with another subsequence, maintaining order.

Example 7. Consider the planar DAG automaton A_p with the following transitions:

$$\varepsilon \xrightarrow{a} (p) \tag{t_1}$$

$$(p) \xrightarrow{b} (p,q) \tag{t_2}$$

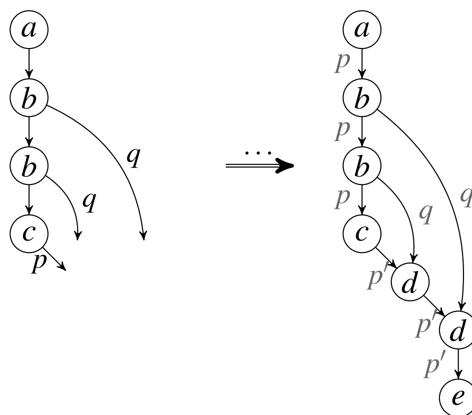
$$(p) \xrightarrow{c} (p') \tag{t_3}$$

$$(p',q) \xrightarrow{d} (p') \tag{t_4}$$

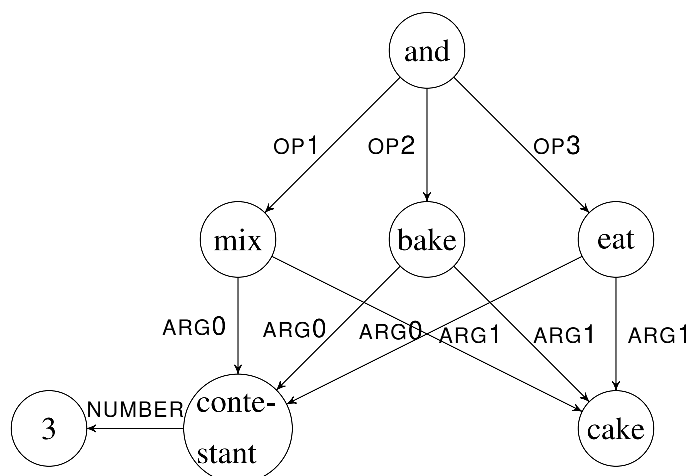
$$(p') \xrightarrow{e} \varepsilon \tag{t_5}$$

In contrast to the non-planar case where two applications of t_2 can produce two unique DAGs, here we can only produce one DAG. The figure below shows a partially and fully derived DAG which can be generated by this planar automaton. The DAG is ordered, where the order of the edges coming out of a node are left-to-right. Therefore, the state p from node c may only combine with the q edge immediately to the right of it, producing the DAG on the right as a result. This automaton is probabilistic using the weight function $w(t_1) = w(t_2) = 1/2$, $w(t_3) = w(t_4) = w(t_5) = 1$.

Our argument in Theorem 1 does not carry across to planar automata since in any frontier of n states, the number of transitions we can apply is linear in n. However, planar DAG automata have other problems that make them unsuitable for modelling meaning representations.



The first problem is that there are natural language constructions that naturally produce non-planar DAGs in AMR. For example, consider the sentence ‘Three contestants mixed, baked and ate a cake’. Its AMR, shown below, is not planar because it has a minor isomorphic to $K_{3,3}$ where the two sets of nodes are: {mix, bake, eat} and {contestant, and, cake}. Any example of coordination where three predicates share two arguments will produce this structure. In the first release of AMR, 117 out of 12,844 graphs are non-planar.



The second problem is that planar DAG automata were defined by Kamimura and Slutzki (1981) to model the derivations of strings using a Turing machine. This seems more expressive than needed to model natural language and it implies that the **emptiness problem**, the decision problem of whether a planar DAG automaton produces an empty language, is undecidable. For non-planar DAG automata, emptiness can be decided in polynomial time (Chiang et al., 2018).

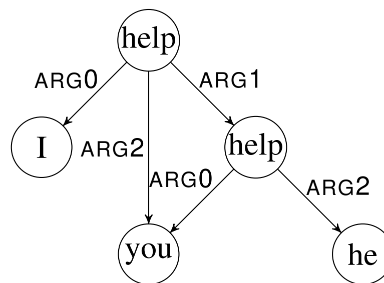
As is shown in Figure 3.1, the planar and non-planar DAG automata languages

are incomparable. We can see that by considering the automata A_{np} in Example 1 and A_p in Example 7. The non-planar language generated by A_{np} contains non-planar DAGs and so clearly this language cannot be generated by any planar DAG automaton. On the other hand, a non-planar DAG automaton could not generate the planar DAG language $\mathcal{L}(A_p)$. To do so would require distinguishing between the q states appearing in the frontier so that the states would be consumed in the correct order to generate only DAGs in $\mathcal{L}(A_p)$. Since there is no bound on the size of these DAGs, there cannot be a finite number of unique states in place of the q states, and so no non-planar DAG automaton could generate this language.

We did not discuss the other families of planar DAG automata here, i.e. those with multiple roots and with unbounded degree. As far as we know, these families have not been studied in detail but we believe it is likely they have similar problems to the planar, single-rooted, bounded degree DAG automata languages.

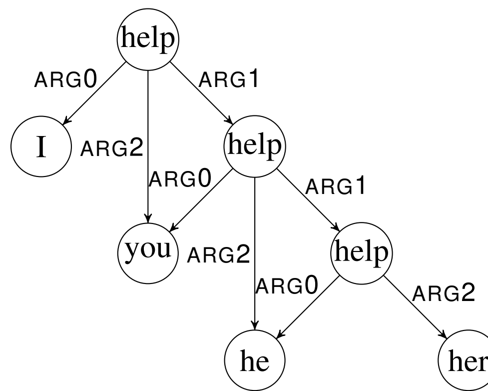
3.3 Implications for semantic DAGs

We introduced DAG automata as a tool for modelling natural language semantics, but the DAG automaton in Theorem 1 is very artificial, so it's natural to ask whether this is a special case with no relevance to natural language. We think this problem is relevant, and give an example. Consider a model of control in a sentence like 'I help you help him' and its simplified AMR shown below:



We can extend the sentence unboundedly with additional helpers, for example this graph represents "I help you help him help her.":

We can continue to do this by adding as many characters as we like. We can write down a DAG automaton that models this phenomenon (for simplicity ignoring the edge



labels):

$$\emptyset \xrightarrow{help} \{p, q', r\} \quad (t''_1)$$

$$\{p\} \xrightarrow{help} \{p, q, r\} \quad (t''_2)$$

$$\{p\} \xrightarrow{help} \{q, r'\} \quad (t''_3)$$

$$\{q'\} \xrightarrow{P} \emptyset \quad (t''_4)$$

$$\{r'\} \xrightarrow{P} \emptyset \quad (t''_5)$$

$$\{q, r\} \xrightarrow{P} \emptyset \quad (t''_6)$$

In the automaton, p stands for a person that can be involved. If the transition t''_2 is applied n times, there are $n!$ ways of combining the n copies of q with the n copies of r in the frontier. As in Theorem 1, this automaton is not probabilistic. Notice that it can produce many more AMRs than the desired ones since it is able to combine any q state with any r state.

	non-planar				planar
bounded degree	yes		no		yes
roots	1	1+	1	1+	1
probabilistic	no	no	no	no	?
decidable emptiness	yes	yes	yes	yes	no
regular paths	no	yes	no	yes	no

Table 3.2: Properties of different families of DAG automata.

3.4 Summary: DAG automata are not suitable for modelling graphs probabilistically

Table 3.2 summarises the properties exhibited by single-rooted, multi-rooted, planar, and unbounded degree automata. It has been argued that all of these properties would be desirable for modelling meaning representations (Drewes, 2017), and so this table suggests that none of these formalisms would suitable for this task.

3.5 Open problems

As we have explored this space, we have identified several open questions, including:

1. Can we identify a property of DAG languages which imply that there is no probabilistic DAG automaton defining them?

2. Is there some subfamily of DAG languages that do generate probabilistic languages?

3. Is it possible to apply the weights in a different way so that a probability distribution is generated?

4. How would we characterise parsing for some "correct" probabilistic automaton?

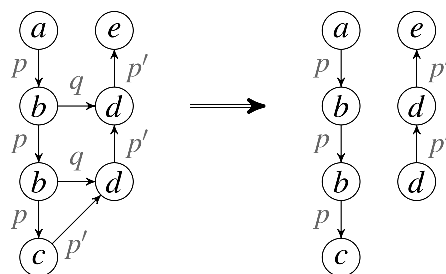
We will discuss these questions in the following.

3.5.1 Properties of DAG languages

This section relates to problems 1 and 2: we explore identifying a testable property of DAG languages that implies that the DAG automata generating them cannot be made probabilistic. This perhaps then will lead to a subfamily of DAG languages that do permit a probability distribution.

Consider the automaton A_{np} in Example 1. The reason that we cannot define a probability distribution with full support is that we end up with $n!$ different DAGs with n nodes and the same weight. This factorial comes from the fact that there is a cyclic transition $\{p\} \xrightarrow{b} \{p,q\}$ in the grammar which has the ability to generate a frontier with an unbounded number of q states. These q states can then be consumed in any order, leading to the factorial.

Take the DAG in Figure 3.3(ii). If you cut (i.e. remove) the edges labelled by q states and the bottom p' edge pointing from c to d then you end up with the disconnected DAG shown below. Applying the same operation to the DAG in Figure 3.4 results in the same disconnected DAG. This resulting DAG contains two connected components. We define the size of this cut to be 3 as we removed 3 edges. We could generalise this to any DAG in this language, where n applications of t_2 give us a cut of size $n+1$. This cut shows us how we can rewire the DAG into a new one, also in the language. We connect c to d, but then each b can connect to each d- this gives us the $n!$ as before.



We consider a specific subclass of non-planar bounded-degree DAG languages—those with a single root and a single leaf. In terms of Figure 3.1, this is a subfamily of (NP,S,BD). The running example automaton A_{np} lies in this family.

Definition 8. *A cut of a DAG* $G = (E, V, lab, src, tar)$ *is a set of edges* $E_c \subseteq E$ *such that:*

- *the subgraph induced by* $E \backslash E_c$ *consists of two connected components, one containing the unique root, and the other the unique leaf,*
- *for all nodes* $v \in V$*, either all of* $e \in in(v)$ *is in* E_c *or in* $E \backslash E_c$ *and all of* $e \in out(v)$ *is in* E_c *or* $E \backslash E_c$*.*

The width of a cut is $|E_c|$. The cut-width of a DAG is the size of its largest cut, and the cut-width of a DAG language is the maximal cut-width of any DAG in its language. Each cut of a DAG corresponds to a possible frontier that can be reached in the derivation of that DAG by some automaton.

Conjecture 1. *Let L be a DAG language with unbounded cut-width. Then there is no automaton that can define a probability distribution with full support over L.*

The idea behind the conjecture is that if L has unbounded cut-width, then any automaton generating L must be able to generate a frontier of unbounded size. This

idea is similar to that of the pumping lemma for regular string languages—if a string is longer than the number of states in the automaton, then we can pump some part of this string to make it as long as we like.

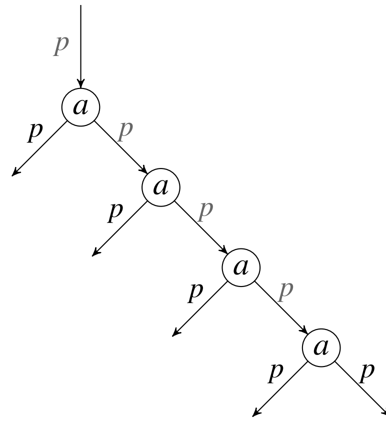
We can consider any DAG automaton as bottom-up or top-down (since here we have a single root and single leaf it is straightforward). The point at which the cut happens in a DAG must then be a frontier which can be reached by from the top-down perspective and the bottom-up perspective. Since the frontier can be unboundedly large, by the pigeon-hole principle there are an unbounded number of copies of some state in the top-down frontier and the bottom-up frontier which need to be matched up. Call this state p. We will explain how if these p states can be distinguished from one another in some way, then there is a factorial number of ways of pairing them up.

We say that we can distinguish two edges from one another if the sequence of states marking the edges on the paths from the root to those edges are not identical. Given a multiset of states S and a state p, we use $\#(S,p)$ to denote the number of copies of p in S. Consider how we end up with a frontier f with some number of p states $\#(f,p)$ which is unbounded. This means that there is some sequence of transitions $T = t_1,\dots,t_n$ that begins at a frontier f_1, ends up at a frontier f_2, and has the property that $\#(f_1,p) < \#(f_2,p)$, i.e. a sequence of transitions that generates more p states than it started with.

If such a sequence of transitions exists, we can use it to distinguish a subset (of unbounded size) of the p state in the frontier from one another. We do this by at each point we apply T, we choose some p state to keep to one side and not be used in any further applications of T. We can always do this since T always generates at least one more p than it consumes. These p states kept aside then will all be distinguishable from one another. For example, we could have the single transition sequence: $\{p\} \xrightarrow{a} \{p,p\}$. We could apply this transition in the way shown in the figure below, leaving the four dangling p edges to the left in the frontier where all can be distinguished from one another by their distance from the root.

We can do this for the top-down generation of the part of the DAG containing the root, and for the bottom-up generation of the part of the DAG containing the leaf. If we have n copies of p that can be distinguished from one another in the frontier, then there are $n!$ ways of wiring them up to one another. As n is unbounded, this generates the divergent sum as in Theorem 1.

As mentioned before the conjecture, we are only dealing here with a specific subclass of DAGs—those with one root and one leaf. There are certainly DAG automata



that cannot be made probabilistic that do not fall into this category. For example, the DAG automaton in Section 3.3 cannot be made probabilistic. We think that some extension of cut-width may be useful to generalise this conjecture, but we do not pursue this here.

3.5.2 Alternative weighting

We now address the way in which we assign weights to DAGs. In Theorem 1, we established that by adding weights to the transitions of a DAG automaton A, we cannot always set these weights in such a way that defines a probability distribution with full support over $\mathcal{L}(A)$. However, we could possibly assign the weights to DAGs in a different way to that done by finite-state string and tree automata. We could do so by, at each point in the derivation, applying a weight to the choice of states that we choose to rewrite. This is not something that we can express using a weighted DAG automaton as they have been defined, but is natural from the perspective of probabilistic modelling in general—each choice we make involves a probability.

Consider the derivation of the DAG in Figure 3.3. The sequence of frontiers during the derivation are:

$$f_1 = \emptyset,$$

$$f_2 = \{p\},$$

$$f_3 = \{p, q\},$$

$$f_4 = \{p, q, q\},$$

$$f_5 = \{p', q, q\},$$

$$f_6 = \{p', q\},$$

$$f_7 = \{p'\},$$

$$f_8 = \emptyset.$$

In moving from f_5 to f_6, either of the q states could have been consumed at this point. If we could factor this choice into the weight of the DAG then we would be able to define a probability distribution. In the general case of n applications of t_2 and t_4, we would like to choose the order in which we consume the q states. In the case of the frontier having a single p' state and n copies of the q state, we could say that there is an equal $1/n$ probability of choosing each q. Then the weight of the language would be:

$$w(L) = \sum_{n=0}^{\infty} [w(t_1)w(t_2)^n w(t_3)n!w(t_4)^n w(t_5)]\, n! = \sum_{n=0}^{\infty} A * B^n * \frac{1}{n!} n! = \sum_{n=0}^{\infty} A * B^n.$$

where $A = w(t_1)w(t_3)w(t_5)$ and $B = w(t_2)w(t_4)$. We can now choose A and B to make $w(L) = 1$.

However, we have missed an important case here in choosing the weight $1/n$ for a frontier with n copies of q and a single p'. At the point of f_5, we could have also chosen to apply t_5 to p'. This would have lead to a frontier of $\{q, q\}$ which is a dead-end and so we cannot derive a DAG from this. If we gave a weight to choosing to apply t_5 at this point, then the model could assign probability mass to derivations that do not end up in the language, possibly leading to an inconsistent probability distribution, i.e. where $w(L) < 1$. Therefore, we would like to assign a weight of 0 to choosing t_5 here. The problem with this is that at the point of being at f_5, how do we know that applying t_5 will lead to a dead end?

The question then is: *Can we decide if applying some transition will lead us down a dead-end path?*

This question is related the reachability problem in Petri nets, which is decidable, but no solution with a primitive recursive running time is known (Reutenauer, 1990; Jones et al., 2012). Drewes and Leroux (2015) showed that it is decidable in polynomial time whether a Petri net is structurally cyclic—a special case of the reachability problem. Chiang et al. (2018) show that this implies that the emptiness problem for multi-rooted non-planar DAG automata is also decidable in polynomial time. In contrast, the emptiness problem for single-rooted non-planar DAG automata is equivalent to the general Petri net reachability problem (Drewes).

Our problem is certainly a form of the Petri net reachability problem, but it is unclear whether we can directly apply the results of Chiang et al. (2018) to show that there is a polynomial algorithm deciding it. Regardless, it also appears that applying a weighting in this way would result in many difficulties when calculating the weight of a DAG given a weighted DAG automaton. To do so, we would need to consider all possible orders of frontiers of states that could have been reached while deriving the DAG. It would also mean that the weight of a derivation of a DAG would not factor over the individual steps made in the derivation—something which is assumed in almost all useful algorithms such as Viterbi, the forward-backward algorithm, and particularly in the algorithms for DAG automata described in Chiang et al. (2018).

3.6 Conclusions

We have shown that by adding weights to the transitions of DAG automata, we cannot always set those weights to generate a useful probability distribution over the language of DAGs that the automaton defines. We have looked at many variants of DAG automata and argued why each is unsuitable for modelling natural language semantics: either they cannot be made probabilistic (non-planar automata), or they are not expressive enough (planar automata).

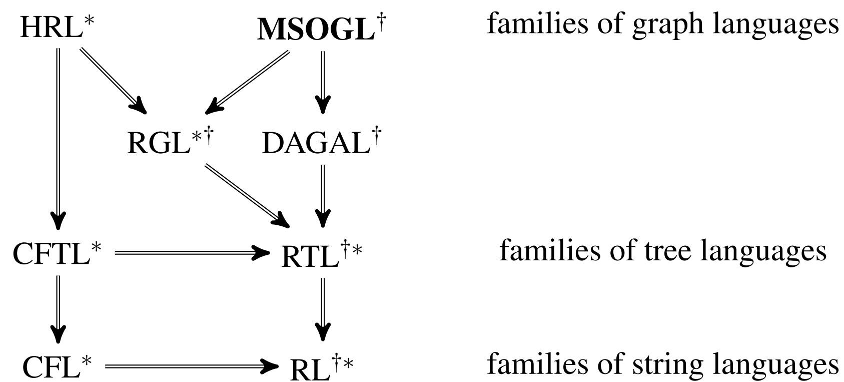
So, although DAG automata are (1) a natural extension of finite-state automata on strings and trees to DAGs, and (2) are closed under intersection, they do not retain the desirable property of being probabilistic in a useful way. By useful, we mean assigning probabilities in a way that could work with algorithms already in use in NLP. In particular, this means that it would be difficult to use dynamic programming algorithms for calculating the weight of a DAG in Chiang et al. (2018) to calculate probabilities. As we discussed in the introduction, we would like to have a model over graphs that is both probabilistic and closed under intersection, therefore DAG automata fall short. We leave this chapter with several open questions which may have answers that could make DAG automata more suited for this task, e.g. if some probabilistic subclass of DAG automata exists, or if there is a better way of assigning weights to the DAGs. For now, however, we believe that DAG automata have many problems which make it difficult for them to model natural language semantics.

As a result, we will look at other ways of extending finite-state string and tree automata to DAGs (or indeed graphs in general). Chapter 4 will deal with one possible extension: the monadic second-order (MSO) graph languages, which contain the DAG

automata languages. This implies that the MSO graph languages may also not be probabilistic. In Chapter 6, we will explore the regular graph grammars (Courcelle, 1991). And in Chapter 7 but also the restricted DAG grammars (Björklund et al., 2016) and the tree-like grammars (Matheja et al., 2015).

Chapter 4

Monadic second-order graph languages



In the previous chapter, we looked at the DAG automata languages. DAG automata are a generalisation of tree and string automata to graphs, as can be seen in the Hasse diagram above. This generalisation is based on the form of the transitions in the automata that generate those languages. Table 3.1 in Chapter 3 summarises the differences between finite-state models on strings, trees, and DAGs.

However, this is not the only way in which we can generalise regular string and tree languages to graphs. We can also look at monadic second-order (MSO) logic, an extension of first-order logic which allows quantification over sets. The languages defined by MSO logic on strings are precisely the regular string languages (Büchi, 1960; Büchi and Elgot, 1958; Trakhtenbrot, 1966), and the languages defined by MSO logic on trees are precisely the regular tree languages (Doner, 1970; Courcelle, 1990). Therefore, another avenue in looking for a "finite-state" model for graphs is to look at MSO logic. MSO logic can operate over different relational structures (Courcelle and

Engelfriet, 2011). The relational structures that we deal with are primarily graphs but we also discuss strings and trees. As we can see in the Hasse diagram at the beginning of the chapter, the DAG automata languages are a subfamily of the MSO graph languages. This suggests that the MSO graph languages may also not be probabilistic. Despite this, we use this chapter to explore MSO and to gain some insight into its useful properties. In Chapter 6 we look at a subfamily of the MSO graph languages called the regular graph languages that are probabilistic.

To get an intuition of how MSO works, we first discuss MSO on strings and show how a finite-state string automaton can be converted into an MSO statement. After that, we will move onto MSO on graphs.

4.1 MSO logic on strings

Consider the string *abc*. We could describe this string using first-order logic by saying something like:

$$\exists a \exists b \exists c(\text{succeeds}(a,b)) \wedge (\text{succeeds}(b,c)).$$

This statement expresses that: there are three letters a, b, c; b succeeds a; and c succeeds b. First-order logic on strings define a proper subset of the regular string languages (McNaughton and Papert, 1971).[1] For example, the language $\{(aa)^n | n \geq 1\}$ cannot be expressed using first-order logic. However, this language is regular and we can use MSO logic to define it (shown below in Example 9).

Rather than operate on letters like in the example above, MSO logic on strings operates on **positions** in a string, where a position corresponds to the gaps between letters including the start and the end. For example, given the string *abc*, we mark the positions using dots as $\cdot a \cdot b \cdot c \cdot$.

MSO logic quantifies over positions and sets of positions but not over predicates. This is in contrast to first-order logic which quantifies over individual positions only, and to second-order logic which can quantify over predicates.[2] Individual positions are denoted as lower case letters, x, and sets of positions are denoted as upper case letters, X. We refer to both x and X as **variables**, and often we refer to the latter as **set variables**. The four **atomic formulas** are:

[1] The languages defined by first-order logic are called the **star-free** languages.

[2] We can say that MSO quantifies over unary predicates since membership of a variable in a set is equivalent to the variable satisfying a unary predicate.

- $x \in X$, which holds if x is in the set X
- $x = y$, which holds if x and y refer to the same position
- $\mathrm{succ}(x,y)$, which holds if y is the successor of x
- $\mathrm{lab}_a(x)$, which holds if the letter immediately following x is a.

To construct a formula, we combine the atomic formulas with the **connectives**: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies); and the **quantifiers**: $\exists$ (there exists), $\forall$ (for all). Let ϕ be an MSO statement and s be a string. Then we say that s **satisfies** ϕ, written as $s \models \phi$, if there is an assignment of variables of ϕ to positions of s that makes ϕ true.

Example 8. Let s be the string $\cdot a \cdot b \cdot c \cdot$ as before. We can always describe a single string in first-order logic, and therefore MSO logic, since we don't need set quantification for a single string. We write the formula for $\cdot a \cdot b \cdot c \cdot$ as follows:

$$\phi : \exists x_1 \exists x_2 \exists x_3 \exists x_4 \Big(\forall x (x = x_1) \vee (x = x_2) \vee (x = x_3) \vee (x = x_4) \tag{4.1}$$

$$\wedge \neg(x_1 = x_2) \wedge \neg(x_1 = x_3) \wedge \neg(x_1 = x_4) \wedge \neg(x_2 = x_3) \wedge \neg(x_2 = x_4) \wedge \neg(x_3 = x_4) \tag{4.2}$$

$$\wedge \mathrm{lab}_a(x_1) \wedge \mathrm{lab}_b(x_2) \wedge \mathrm{lab}_c(x_3) \wedge \mathrm{succ}(x_1,x_2) \wedge \mathrm{succ}(x_2,x_3) \wedge \mathrm{succ}(x_3,x_4)\Big). \tag{4.3}$$

Lines 4.1 and 4.2 specify that there are exactly four positions in the string. Line 4.3 then specifies the labels and orders of the positions. By assigning the positions of the string $\cdot a \cdot b \cdot c \cdot$ to x_1, x_2, x_3, x_4 by moving left-to-right, the formula holds and so we can say that $s \models \phi$.

We can use an MSO formula ϕ over an alphabet A to define a language by saying $\mathcal{L}(\phi) = \{s \in A^* \mid s \models \phi\}$, i.e. the language is made up by exactly the set of strings that satisfy ϕ. We say that a string language L is **MSO-definable** if there exists an MSO formula ϕ such that $L = \mathcal{L}(\phi)$.

Example 9. We now show how we can use MSO logic to define a language, namely the language $(aa)^n$ which cannot be expressed in first-order logic. We define two sets, Y_o and Y_e and we assign each of the positions of the string to these sets, Y_o for odd positions and Y_e for even positions. The formula is as follows:

$$\exists Y_o \exists Y_e \Big(\forall x\big((x \in Y_o \vee x \in Y_e) \wedge \neg(x \in Y_o \wedge x \in Y_e)\big) \tag{4.4}$$

$$\forall x \in Y_o \big((\exists y.\mathrm{succ}(x,y) \Rightarrow \mathrm{lab}_a(x) \wedge y \in Y_e) \wedge (\exists y.\mathrm{succ}(y,x) \Rightarrow y \in Y_e)\big) \tag{4.5}$$

$$\wedge \forall x \in Y_e \Big(\mathrm{lab}_a(x) \wedge \big((\exists y \in Y_o.\mathrm{succ}(x,y)) \wedge (\exists y \in Y_o.\mathrm{succ}(y,x))\big)\Big) \tag{4.6}$$

Line 4.4 says that each position in the string is assigned to exactly one of the sets Y_o and Y_e. Line 4.5 says that all neighbours of a position in Y_o lie in Y_e, and that each position besides the last is labelled by an a (since the last position has no label). Similarly, Line 4.6 says that all neighbours of a position in Y_e lie in Y_o and all have the label a.

Since the string languages defined by MSO logic are equivalent to the regular string languages (Büchi, 1960; Büchi and Elgot, 1958; Trakhtenbrot, 1966), for each finite-state string automaton A, there is a corresponding MSO formula ϕ such that $\mathcal{L}(A) = \mathcal{L}(\phi)$ and vice-versa.

We walk through an example of how to convert a finite-state automaton into an MSO formula to convey the main intuitions of why they are equivalent. Recall the definition of a run of a DAG automaton from the previous chapter. There, a run ρ over a DAG G assigned each edge in G to a state in the automaton. This idea also applies to using finite-state automata to recognise strings—the run assigns positions in the string to states. Converting a finite-state automaton into an MSO statement essentially involves constructing a statement which simulates the run of the automaton over a string: it first guesses some assignment of positions to states, and then it checks that assignment is valid under the definition of the automaton.

Throughout the thesis, we will define MSO formulas that we will re-use. When we do so, we will use small capitals for their definitions, for example we will now define the formula $\textsc{Part}_n(X_1,\ldots,X_n)$ to express that the n sets of positions $X_1,\ldots,X_n$ partition the positions in a string. It is written as $\textsc{Part}_n(X_1,\ldots,X_n)$:

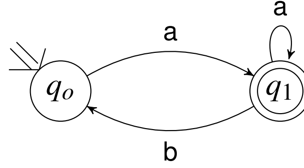
$$\forall x\,(x \in X_1 \cup \cdots \cup X_n) \wedge (\neg(x \in X_1 \cap X_2) \wedge \neg(x \in X_1 \cap X_3) \wedge \cdots \wedge \neg(x \in X_{n-1} \cap X_n))\,. \tag{4.7}$$

This formula says that each position x in the string is assigned to at least one state $X_1,\ldots,X_n$ and that the assignment is mutually exclusive—no position can be assigned to more than one state. The partition formula will be used many times throughout this thesis, note that we already used it in Equation 4.4.

In a string, there is always a unique starting position and a unique ending position. We identify these using the formulas $\textsc{First}(x)$ and $\textsc{Last}(x)$ as follows:

$$\textsc{First}(x) : \forall y \neg \mathrm{succ}(y,x), \tag{4.8}$$

$$\textsc{Last}(x) : \forall y \neg \mathrm{succ}(x,y). \tag{4.9}$$



These formulas say that the first position has no predecessor and the last position has no successor.

Example 10. Let A be the automaton shown above. We can encode all of the constraints of A into MSO logic. For each state in A, we define a set variable in MSO logic. Therefore, since we have states q_0 and q_1, we need corresponding set variables X_0 and X_1. We need to specify the start state, final state(s), and the transitions. Given a transition of the form (q_i, a, q_j) we can encode this as: $x \in X_i \wedge \text{lab}_a(x) \Rightarrow \exists y \left(s(x,y) \wedge y \in X_j\right)$. Let $\textsc{Aut}_A$ be the MSO formula corresponding to the automaton A:

$$\exists X_0 \exists X_1 \textsc{Part}_2(X_1, X_2) \tag{4.10}$$

$$\wedge \forall x \textsc{First}(x) \Rightarrow x \in X_0 \tag{4.11}$$

$$\wedge \forall x \textsc{Last}(x) \Rightarrow x \in X_1 \tag{4.12}$$

$$\wedge \forall x (x \in X_0 \wedge \text{lab}_a(x)) \Rightarrow \exists y (\text{succ}(x,y) \wedge y \in X_1) \tag{4.13}$$

$$\wedge \forall x (x \in X_1 \wedge \text{lab}_a(x)) \Rightarrow \exists y (\text{succ}(x,y) \wedge y \in X_1) \tag{4.14}$$

$$\wedge \forall x (x \in X_1 \wedge \text{lab}_b(x)) \Rightarrow \exists y (\text{succ}(x,y) \wedge y \in X_0). \tag{4.15}$$

Line 4.10 constrains that every position must be in exactly one state. Line 4.11 defines q_0 as the start state and Line 4.12 defines q_1 as the only final state. Line 4.13 defines the transition from q_0 to q_1, Line 4.14 defines the transition from q_1 to itself, and Line 4.15 defines the transition from q_1 to q_0.

For any string s that can be recognised by the automaton A, we can say that $s \models \textsc{Aut}_A$. For example, take the string $\cdot a \cdot a \cdot b \cdot a$ and assume the positions are $1,2,3,4,5$ numbered left-to-right. Assign positions 1 and 5 to X_0 and positions $2,3$ and 4 to X_1. Then the formula $\textsc{Aut}_A$ is true.

Although the automaton in the example is deterministic, we can also express non-determinism in MSO logic. We do so by using disjunctions. For example, if we had an extra state q_2 (and corresponding set variable X_2) and a transition labelled a from q_0 to q_2 then instead of Line 4.13 we would have:

$$\wedge \forall x (x \in X_0 \wedge \text{lab}_a(x)) \Rightarrow \exists y (\text{succ}(x,y) \wedge (y \in X_1 \vee y \in X_2)).$$

Similarly, we can express that there are multiple final states by adding a disjunction to Line 4.12.

The formula AUT_A has no unbound (or free) variables while the formula $\text{First}(x)$ has unbound variable x. For formulas of the form $\phi(\mathcal{W})$ where $\mathcal{W}$ is a collection of variables, we refer this set of variables as the **parameters** of ϕ. We can talk about a string satisfying a formula with unbound variables when we pair the string with a function which maps the unbound variables to positions in the string. For example, let s be a string and $\phi(\mathcal{W})$ be a formula with parameters $\mathcal{W}$. Let α be a function from $\mathcal{W}$ to positions in s, we refer to α as a **parameter assignment**. Then we say that $(s,\alpha) \models \phi(\mathcal{W})$ if there is some assignment of bound variables in $\phi(\mathcal{W})$ to positions in s along with α that satisfies ϕ.

Example 11. Take the formula $\text{FIRST}(x) : \forall y \neg s(y,x)$. Then x is unbound in FIRST and y is bound in FIRST. Take the string $s = \cdot a \cdot a \cdot b \cdot a \cdot$ and number the positions as usual from left to right. Let α assign x to position 1, then $(s,\alpha) \models \text{FIRST}(x)$ as with x assigned to 1, for all other positions y, the successor of y is not x.

4.2 MSO languages are closed under intersection

It is straightforward to see why MSO languages are closed under intersection. Let L_1 and L_2 be MSO-definable languages, then there exist MSO statements ϕ_1 and ϕ_2 such that $L_1 = \mathcal{L}(\phi_1)$ and $L_2 = \mathcal{L}(\phi_2)$. Then

$$L_1 \cap L_2 = \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2) = \mathcal{L}(\phi_1 \wedge \phi_2).$$

4.3 MSO on graphs

Now that we have established how MSO operates over strings, we can see how it extends to graphs.[3] Graphs were defined in Definition 1 in Chapter 2.

There are two main varieties of MSO on graphs, one where we quantify only over nodes (called MS_1) and one where we quantify over both nodes and edges (called MS_2). We say that the **domain** of MS_1 is nodes and the domain of MS_2 is nodes and edges. We will describe both here, giving examples of what each can express.

[3]The definitions of MSO and MSO transducers appearing in the following sections have been adapted from Courcelle and Engelfriet (2011).

4.3.1 MS_1

MS_1 refers to monadic second-order logic over graphs which quantifies only over the nodes in the graph. The atomic formulas are:

- $x \in X$, indicating the node x is in set X,
- $x = y$ indicating that x and y refer to the same node,
- $\text{lab}_a(x)$, indicating the node x has label a,
- $\text{edge}^1_a(x,y)$, indicating there is an edge with label a that connects nodes x and y (the superscript 1 is used to indicate that we are using MS_1).

Note that the notation here differs slightly from that introduced in Chapter 2 but it captures the same ideas. There we used $\text{src}(e)$ and $\text{tar}(e)$ to specify the endpoints of an edge; and lab_e to specify the label. In MS_1 we do not quantify over edges so we represent an edge as the predicate $\text{edge}^1_a(x,y)$, which is true if there is an a-labelled edge between nodes x and y. We can use MS_1 to represent both directed and undirected graphs. In the directed case, $\text{edge}^1_a(x,y)$ means there is an edge from node x to node y.

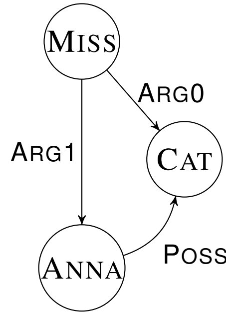


Figure 4.1: The graph representing a simplified AMR of the sentence “Anna’s cat misses her”.

Example 12. Let G be the graph in Figure 4.1 which is repeated from the introduction. We write down the MS_1 describing this graph. We use the variables x for the node labelled by ANNA, y for CAT, and f for MISS.

$$\exists x \exists y \exists f \Big(\text{lab}_{\text{ANNA}}(x) \wedge \text{lab}_{\text{CAT}}(y) \wedge \text{lab}_{\text{MISS}}(f)$$
$$\wedge \text{edge}^1_{\text{ARG1}}(f,x) \wedge \text{edge}^1_{\text{ARG0}}(f,y) \wedge \text{edge}^1_{\text{POSS}}(x,y) \Big)$$

Although the graph satisfies this MSO statement, the statement defines many other graphs. To make the statement exactly correspond to this graph, we would also need to stipulate

- that the nodes x, y and f are unique:

$$\neg(x = y) \wedge \neg(x = f) \wedge \neg(y = f);$$

- that there are no other nodes in the graph:

$$\forall v(v = x \vee v = y \vee v = f);$$

- and that there are no other edges:

$$\forall v \forall v' \text{edge}_{\text{ARG1}}(v, v') \Rightarrow (v = f \wedge v' = y)$$
$$\forall v \forall v' \text{edge}_{\text{ARG0}}(v, v') \Rightarrow (v = f \wedge v' = x)$$
$$\forall v \forall v' \text{edge}_{\text{POSS}}(v, v') \Rightarrow (v = x \wedge v' = y)$$

where we assume that the edge label set is $\{\text{ARG0}, \text{ARG1}, \text{POSS}\}$.

Example 13. We can use MS_1 to express that a graph is bipartite.

Definition 9. *A **bipartite graph** is a graph whose nodes can be partitioned into two disjoint subsets such that each edge in the graph has its two endpoints in opposite sets.*

We can write a statement BP that any bipartite graph will satisfy using using MS_1 as follows:

$$\exists X \exists Y \text{PART}_2(X,Y) \left(\forall x \forall y \text{edge}^1(x,y) \vee \text{edge}^1(y,x) \Rightarrow (x \in X \wedge y \in Y) \vee (x \in Y \wedge y \in X)\right) \tag{4.16}$$

Where $\text{edge}^1(x,y) = \bigvee_{\gamma \in \Gamma} \text{edge}^1_\gamma(x,y)$ says there is some edge between x and y without specifying the label.

4.3.2 MS_2

The second type of MSO logic on graphs is MS_2, whose domain includes both nodes and edges. The atomic formulas are written as:

- $x \in X$, indicating the node or edge x is in set X,
- $x = y$ indicating that x and y refer to the same node or edge,
- $\mathrm{lab}_a(x)$, indicating the node or edge x has label a,
- $\mathrm{edge}^2(e,x,y)$ indicating there is an edge e which connecting node x and node y.

Again here, if we want to use directed graphs, $\mathrm{edge}^2(e,x,y)$ represents an edge going from x to y.

Example 14. Returning to the graph in Figure 4.1, we can now write down an MS_2 formula that this graph satisfies.

$$\exists x \exists y \exists f \exists e_0 \exists e_1 \exists e_p \Big(\mathrm{lab}_{\textsc{Anna}}(x) \wedge \mathrm{lab}_{\textsc{Cat}}(y) \wedge \mathrm{lab}_{\textsc{Miss}}(f) \\ \wedge \mathrm{lab}_{\textsc{Arg0}}(e_0) \wedge \mathrm{lab}_{\textsc{Arg1}}(e_1) \wedge \mathrm{lab}_{\textsc{Poss}}(e_p) \\ \wedge \mathrm{edge}^2(e_0,f,y) \wedge \mathrm{edge}^2(e_1,f,x) \wedge \mathrm{edge}^2(e_p,x,y) \Big)$$

The representation implied by MS_2 is often called an **incidence** graph. Courcelle and Engelfriet (2011) refer to a graph G's representation in MS_1 as $\lfloor G \rfloor$ and in MS_2 as $\lceil G \rceil$. The difference between the two is that given a graph with multiple edges with the same label between some pair of nodes, MS_1 will only express that there is an edge between the nodes with that label whereas MS_2 will be able to identify each edge that connects the pair.

The graph languages defined by MS_1 are a subfamily of the graph languages defined by MS_2. To display this point, we walk through the MS_2 statement expressing that a graph is Hamiltonian—something which MS_1 cannot express.

MS_2 can express that a graph is **Hamiltonian**, which means that there is an undirected path through the graph which begins and ends at the same node and the path visits each other node in the graph exactly once (called a Hamiltonian cycle). To establish that a graph is Hamiltonian, we need to find the set of edges that lie on the path, which we call E_H here. We can express that a variable refers to an edge by checking if it lies in the edge position of the edge predicate, i.e. $\mathrm{edge}^2(e,x,y)$ implies that e is an edge and x and y are nodes.

$$\textsc{Edge}(e) : \exists x \exists y \mathrm{edge}^2(e,x,y)$$

We can then define NODE as:

$$\text{NODE}(x) : \neg\text{EDGE}(x), \tag{4.17}$$

and we can express that the set N contains exactly the set of all nodes with:

$$\text{NODES}(N) : ((\forall n(n \in N) \Rightarrow \text{NODES}(n)) \wedge \forall n(\text{NODES}(n) \Rightarrow (n \in N)))$$

These definitions imply that an object which is floating and is not attached to anything else is always a node. However, since we deal with connected graphs in this thesis, this case should not arise.

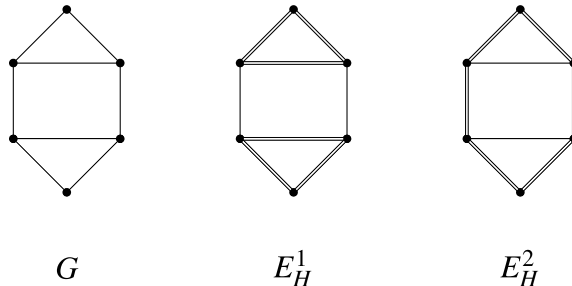
It is useful to say that a node is incident to some edge without needing to name the other node the edge is attached to. We define the formula INC:

$$\text{INC}(e,x) : \exists y \left(\text{edge}^2(e,x,y) \vee \text{edge}^2(e,y,x)\right).$$

For E_H to define a Hamiltonian cycle, we need each node in N to be incident to exactly two edges in E_H:

$$\text{N2E}(E_H,N) : \forall n(n \in N \Rightarrow \exists e_1 \in E_H \exists e_2 \in E_H(\neg(e_1 = e_2) \wedge \text{INC}(e_1,n) \wedge \text{INC}(e_2,n) \\ \wedge \forall e_3(e_3 \in E_H \wedge \text{INC}(e_3,n) \Rightarrow (e_3 = e_1) \vee (e_3 = e_2))). \tag{4.18}$$

At this point, E_H could describe a collection of cycles but not one continuous cycle through the graph. For example, take the graph G below shown on the left. E_H^1 shows a set of edges (shown as double lines) which satisfy all of the requirements so far but do not make a Hamiltonian cycle since it is made up of two cycles. E_H^2 shows a Hamiltonian cycle of G.



G $\quad E_H^1 \quad E_H^2$

To express that it is one cycle, we need to say that for any non-empty proper subset N' of the set of nodes, there must be some edge in E_H which has exactly one endpoint in N'.

$$\text{1CYC}(E_H,N) : \forall N'((\exists n \in N') \wedge (\forall n(n \in N') \Rightarrow n \in N) \wedge (\exists n(n \in N) \wedge \neg(n \in N')) \\ \Rightarrow \exists e \in E_H \exists n_1 \exists n_2(\text{INC}(e,n_1)) \wedge (\text{INC}(e,n_2)) \wedge (n_1 \in N') \wedge \neg(n_2 \in N') \tag{4.19}$$

Consider E_H^1 in the figure above. If we removed the top three nodes then the formula $1\textsc{cyc}$ would fail as there would be no edge in E_H^1 that has only one endpoint in the remaining nodes. E_H^2, however, would pass $1\textsc{cyc}$ since both of the side edges would have only endpoints on the lower nodes.

We then combine these formulas together to get the formula expressing the existence of a Hamiltonian cycle in a graph:

$$\exists E_H \exists N \Big((\forall e(e \in E_H) \Rightarrow \textsc{Edge}(e)) \wedge \textsc{Nodes}(N) \wedge \textsc{n2e}(E_H, N) \wedge 1\textsc{cyc}(E_H, N) \Big).$$

MS_1, however, cannot express that a graph is Hamiltonian (Courcelle and Engelfriet, 2011). The proof shows that there is an MS_1 statement defining a Hamiltonian cycle if and only if there is an MSO statement on strings which can describe the language containing an equal number of as and bs—a language which is not regular. It does so by taking each string s in $\{a,b\}^*$ and constructing a corresponding complete bipartite graph. The bipartite graph has one set of nodes for each a appearing in s and another set of nodes for each b appearing in s. Then each a node is connected to each b node. This graph has a Hamiltonian cycle if and only if s has an equal number of as and bs.

Therefore, there are MS_2 languages that are not MS_1 languages. In the other direction, for each MS_1 statement, we can convert it into an MS_2 statement by saying that each variable quantified over in the MS_1 statement must be a node in the MS_2 statement and at each point in the MS_1 statement that referred to an edge, we convert $\text{edge}_a^1(x,y)$ into $\exists! e \text{edge}^2(e,x,y) \wedge \text{lab}_a(e)$ where ! denotes that this e is unique. Therefore, the MS_1 languages are a proper subfamily of the MS_2 languages.

While they have their differences, an important property common to both is that neither can express the cardinality of a set. For example there is no MSO statement expressing that $|X| = |Y|$ for set variables X and Y. There is a variant on MSO called counting MSO which allows adds a predicate $\text{Card}_{p,q}(X)$ expressing that $|X| \mod p \equiv q$, but this is beyond the scope of this thesis. For more detail on counting MSO, refer to Courcelle and Engelfriet (2011).

4.4 MSO transductions

In this section, we introduce MSO transductions. They can be used to prove that a language is MSO-definable, as we will show in Chapter 6. MSO transductions are a generalisation of finite-state string and tree transductions. MSO transducers can map

within and also between MS_1 and MS_2. A transducer that maps from MS_1 to MS_2 is called an $MS_{1,2}$ transducer. We first define $MS_{1,1}$ transducers and give some examples.

Let Σ be a finite set of node labels and Γ be a finite set of edge labels, then $G_{\Sigma,\Gamma}$ is the set of all graphs over those sets of labels. This can be seen as the graph analogy to A^* being the set of all strings over the alphabet A.

Definition 10. *An $MS_{1,1}$ transducer $\tau : G_{\Sigma,\Gamma} \to G_{\Sigma',\Gamma'}$ is defined by the definition scheme:*

$$\tau = \langle \tau\text{-}\rho, \tau\text{-}node(x), ((\tau\text{-}edge^1_\gamma(x,y))_{\gamma\in\Gamma'}, (\tau\text{-}lab_\sigma)_{\sigma\in\Sigma'}) \rangle.$$

*τ-ρ is an MS_1 formula called the **precondition** which input graphs must satisfy; τ-$node(x)$ is the **domain formula** defining the output nodes; τ-$edge^1_\gamma(x,y)$ for each γ define the output edges; and τ-$lab_\sigma(x)$ define the output node labels. Collectively, the edge and label formulas are known as the **relation formulas**.*

Example 15. The following $MS_{1,1}$ transducer τ produces the edge-complement of an unlabelled undirected input graph,

$$\begin{aligned} &\tau\text{-}\rho : \text{True},\\ &\tau\text{-node}(x) : \text{True},\\ &\tau\text{-edge}^1(x,y) : \neg(x=y) \wedge \neg\text{edge}^1(x,y). \end{aligned} \tag{4.20}$$

τ-ρ tells us that any input graph is allowed, τ-node tells us that each node in the input graph should be in the output graph, and τ-edge$^1(x,y)$ tells us to only have an edge between x and y in the output if they are different and did not have an edge between them in the input graph. Figure 4.2 shows an input and output of this transducer.

We could alter τ by adding the precondition BP expressing that the input graph must be bipartite using the formula from Equation 4.16. This defines τ_{BP}:

$$\begin{aligned} &\tau_{BP}\text{-}\rho : \text{BP},\\ &\tau_{BP}\text{-node}(x) : \text{True},\\ &\tau_{BP}\text{-edge}^1(x,y) : \neg(x=y) \wedge \neg\text{edge}^1(x,y). \end{aligned} \tag{4.21}$$

Since the input in Figure 4.2 is bipartite, the output of τ_{BP} would be the same for this graph as it was for τ. However, if the input was as in Figure 4.3, there is no output generated since the input is not bipartite.

Figure 4.2: An input and output to the MSO transducers τ and τ_{BP} defined in Equations 4.20 and 4.21.

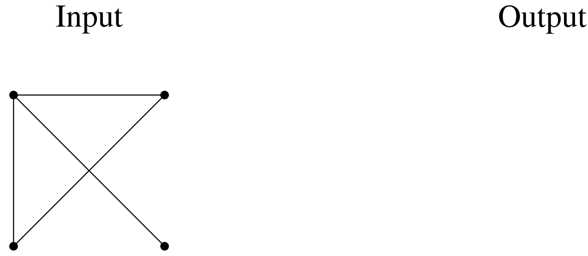


Figure 4.3: An input and output to the MSO transducer τ_{BP} defined in Equation 4.20. No output is generated since the input does not pass the precondition.

We could instead have altered τ by changing the domain formula to say that the nodes must have degree less than 3. The MS_1 formula specifying that a node in an undirected graph has degree less than 3 is:

$$\mathrm{deg}_{<3}(x) : \neg\Big(\exists y_1 \exists y_2 \exists y_3 \Big(\neg\big((y_1 = y_2) \vee (y_1 = y_3) \vee (y_2 = y_3)\big) \\ \wedge \mathrm{edge}^1(x, y_1) \wedge \mathrm{edge}^1(x, y_2) \wedge \mathrm{edge}^1(x, y_3)\Big)\Big).$$

Now we define the new transducer τ_{deg}:

$$\begin{aligned} &\tau_{\mathrm{deg}}\text{-}\rho : \mathrm{True}, \\ &\tau_{\mathrm{deg}}\text{-node}(x) : \mathrm{deg}_{<3}(x), \\ &\tau_{\mathrm{deg}}\text{-edge}^1(x,y) : \neg(x = y) \wedge \neg\mathrm{edge}^1(x,y). \end{aligned} \tag{4.22}$$

An input and output of τ_{deg} is shown in Figure 4.4. The top left node is not present in the output since it fails the domain formula τ_{deg}-node(x). The edges of the output are then only generated between the nodes that passed the domain formula.

The above examples dealt only with $\mathrm{MS}_{1,1}$ transducers—transducers that map from MS_1 graphs to MS_1 graphs. We now see how to generalise this idea. To define an MSO

Figure 4.4: An input and output to the MSO transducer τ_{deg} defined in 4.20.

transducer which maps *from* MS_2, then the formulas for the precondition, domain, and relations must be written in MS_2. To define an MSO transducer τ which maps *into* MS_2, the domain formulas have the form:

$$\tau\text{-n/e}(x)$$

where n/e stands for node/edge; and the relation formulas have the form:

$$\tau\text{-edge}^2(e,x,y),$$

and

$$\tau\text{-lab}_a(x)$$

where $a \in \Sigma' \cup \Gamma'$.

The differences between the various types of transducer (e.g. $MS_{1,2}$ vs $MS_{1,1}$) is beyond the scope of this work but there is a detailed study in Courcelle and Engelfriet (2011).

Example 16. For example, we could write the transducer τ from Equation 4.20 as an $MS_{2,1}$ transducer:

$$\tau\text{-}\rho : \text{True}$$
$$\tau\text{-node}(x) : \textsc{Node}(x)$$
$$\tau\text{-edge}^1(x,y) : \neg(x = y) \wedge \neg\exists e(\text{edge}^2(e,x,y))$$

where $\textsc{Node}(x)$ is as defined in Equation 4.17.

In general, MSO transducers use parameters in their definition. We write an $MS_{1,1}$ transducer with parameters as:

$$\tau(\mathcal{W}) : \langle \tau\text{-}\rho(\mathcal{W}), \tau\text{-node}(x, \mathcal{W}), ((\tau\text{-edge}^1_\gamma(x,y,\mathcal{W}))_{\gamma\in\Gamma'}, (\tau\text{-lab}_\sigma(x,\mathcal{W})_{\sigma\in\Sigma'})\rangle$$

These parameters take the form of variables and sets of variables. For each valid mapping of domain elements to these parameters, we generate an output of the transducer. The role of parameters in MSO transducers is to allow nondeterminism. Given a graph G and a parameter assignment α from $\mathcal{W}$ to $V_G \cup E_G$ such that $(G,\alpha) \models \tau$-$\rho(\mathcal{W})$, we define the output of the $\mathrm{MS}_{1,1}$ transducer $\tau(G,\alpha) = (N, E, \mathrm{lab})$ such that N is the set of output nodes defined as:

$$\{x \mid (G,x,\alpha) \models \tau\text{-node}(x,\mathcal{W})\}.$$

E is the set of output edges defined as:

$$\{\mathrm{edge}^1_\gamma(x,y) \mid (G,x,y,\alpha) \models \tau\text{-edge}^1_\gamma(x,y,\mathcal{W})\}$$

and lab is the set of output labels defined as:

$$\{\mathrm{lab}_\sigma(x) \mid (G,x,\alpha) \models \tau\text{-lab}_\sigma(x,\mathcal{W})\}$$

Define

$$\tau(G) = \{\tau(G,\alpha) \mid (G,\alpha) \models \rho(\mathcal{W})\}$$

for an individual graph G. And for a language L,

$$\tau(L) = \{\tau(G) \mid G \in L\}.$$

Example 17. We define a simple $\mathrm{MS}_{1,1}$ transducer τ that uses a parameter $\mathcal{W}$. For simplicity, we assume the edges are unlabelled. We first define the precondition which specifies how we assign nodes to $\mathcal{W}$:

$$\tau\text{-}\rho(\mathcal{W}) : \exists x \in \mathcal{W} \exists y \in \mathcal{W} \left(\mathrm{edge}^1(x,y) \wedge \forall z \in \mathcal{W} \, (z = x \vee z = y)\right).$$

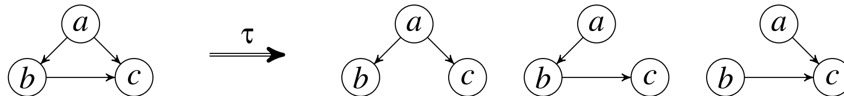
In other words, we require the graph to have at least two nodes with an edge between them. The parameter $\mathcal{W}$ captures a single pair of nodes that satisfy this requirement, and only that pair of nodes—so in general, the number of valid assignments to $\mathcal{W}$ will match the number of edges in the input graph. Then we define the output domain and the relation formulas:

$$\tau\text{-node}(x,\mathcal{W}) : \mathrm{True},$$

$$\tau\text{-edge}^1(x,y,\mathcal{W}) : \mathrm{edge}^1(x,y) \wedge \neg(x \in \mathcal{W} \wedge y \in \mathcal{W}),$$

$$\tau\text{-lab}_a(x) : \mathrm{lab}_a(x).$$

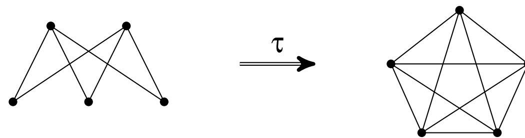
The domain formula just tells us to copy everything over from the input graph and the labels also stay the same. The edge formula says to copy all of the edges from the input graph except any edge nodes that lie in $\mathcal{W}$. For each valid choice of x and y, we will get a different output graph. An input with its outputs can be seen in the figure below. In the first case, $x = b$ and $y = c$, in the second case, $x = a$ and $y = c$, and in the third case $x = a$ and $y = b$.



MSO transducers can be used to show that a language is MSO-definable via the backwards translation theorem (Courcelle and Engelfriet, 2011). The theorem is a generalisation to graphs of the fact that regular string and tree languages are closed under inverse finite-state transductions (Hopcroft and Ullman, 1979; Comon et al., 2007). In the next chapter, we will use MSO transducers to show that the regular graph languages (Courcelle, 1991) are MSO-definable.

Theorem 2 (Backwards Translation Theorem)**.** *If L is an MSO-definable graph language and f is an MSO graph transduction then $f^{-1}(L) = \{G \mid f(G) \in L\}$ is effectively MSO-definable.*

We show an example of the backwards translation theorem using a transducer that maps from bipartite graphs to complete graphs.



Example 18. We can write an $\text{MS}_{1,1}$ transducer over undirected unlabelled graphs τ_{bc} which takes any bipartite graph as input and outputs a complete undirected graph over the same nodes. The figure above shows an example input and output of this transducer. This is defined as:

τ_{bc}-ρ : BP

τ_{bc}-node(x) : True

$\tau_{\text{bc}}\text{-edge}^1(x,y)$: True

where BP is defined in Equation 4.16. Let L_C be the language of complete undirected graphs without edge or node labels. We can define L_C using the MS_1 formula:

$$\forall x \forall y(\text{edge}^1(x,y)).$$

Using the backwards translation theorem we know that since L_C is MSO-definable, $\tau_{\text{bc}}^{-1}(L_C)$ is also MSO-definable. Therefore,

$$\{G \mid \tau_{\text{bc}}(G) \in L_C\}$$

is MSO-definable. Since the input to τ_{bc} must be bipartite (by τ-ρ), we know that $\tau_{\text{bc}}^{-1}(L_C)$ is the set of all undirected bipartite graphs without node or edge labels. We already know that this is MSO-definable, since we have the formula BP in Equation 4.16. Therefore, the backwards translation theorem holds here.

Given a finite-state transducer over strings, if the input language is regular then the output language is also regular. This does not hold for MSO transducers: if L is MSO-definable, and τ is an MSO transducer, then $\tau(L)$ may not be MSO-definable. We will now give an example of an MSO transducer that takes an MSO-definable language as input and generates a language that is not MSO-definable.

Example 19. It is possible to define an MSO transducer that copies each input element, finitely many times. The definition we previously defined does not allow this and that type of transducer is referred to as a non-copying transducer. We may only make some number k copies of each input. To allow copying in an $\text{MS}_{1,1}$ transducer, we define the domain of the output to be

$$N = \cup_{i \in [k]} \{(x,i) \mid (G,x) \models \text{NODE}(x)\}.$$

Each of the nodes in the output is written (x,i) to indicate that this is the ith copy of x. This means now that, in the case of an $\text{MS}_{1,1}$ transducer, our output domain is over pairs of nodes and integers. The following is an example of a 2-copying $\text{MS}_{1,1}$ transducer τ:

τ-ρ : True,

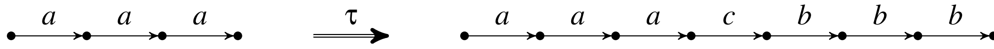
$\tau\text{-node}(x)$: True,

$\tau\text{-edge}^1_a((x,1),(y,1)) : \text{edge}^1_a(x,y),$

$\tau\text{-edge}^1_b((x,2),(y,2)) : \text{edge}^1_a(x,y).$

$\tau\text{-edge}^1_c((x,1),(y,2)) : \neg\exists z(\text{edge}^1(x,z)) \wedge \neg\exists z(\text{edge}^1(z,y)).$

Let the input to the transducer be a graph consisting of a chain of n a-labelled edges (as shown in the left-hand side of the figure below). Then the output will consist of a chain of $2n+1$ edges: the first n edges labelled a, then an edge labelled c, and the final n edges labelled b (as shown in the right-hand side of the figure below). Consider the language is graphs of this type with lengths in $n \in \mathbb{N}$ and call this L_{a^*}. Since this transformation is over chain graphs, we can represent it as a string transformation from the language a^* to the language $a^n cb^n$. The input language is a regular language and so is MSO-definable. However, the output language is not regular and so is not MSO-definable. Therefore, unlike finite-state string and tree transducers, MSO transducers do not preserve MSO-definability.



To summarise, given languages L, L', an MSO transducer τ, and $L = \tau^{-1}(L')$; if L' is MSO-definable then L is, but if L is MSO-definable then L' may not be.

4.5 MSO and DAG automata

All DAG automata languages are MSO-definable. We can convert a non-planar DAG automaton $A = (Q, \Sigma, R)$ into an MS_2 statement in a similar way to how we can convert an FSA into MSO. Recall how the MSO statement for an FSA simulated a run of the automaton. The MSO statement for DAG automata uses set variables to guess an assignment of each edge in a DAG to a state, and then the formula checks if that assignment is valid.

For each state in $q \in Q$, we define a set variable X_q. We require that the X_q sets partition the edges:

$$\begin{aligned}
&\text{E-PART}(X_{q_1}, \ldots, X_{q_{|Q|}}) : \\
&\qquad \forall x \Big((\text{EDGE}(x) \Rightarrow (x \in X_{q_1}) \vee \cdots \vee (x \in X_{q_{|Q|}})) \\
&\qquad \wedge \neg(x \in X_{q_1} \cap X_{q_2}) \wedge \cdots \wedge \neg(x \in X_{q_{|Q|-1}} \cap X_{q_{|Q|}}) \\
&\qquad \wedge ((x \in X_{q_1}) \vee \cdots \vee (x \in X_{q_{|Q|}})) \Rightarrow \text{EDGE}(x) \Big)
\end{aligned}$$

To check that the assignment is valid, we look at each node x in the DAG and check that there is some transition $\{p_1,\dots,p_n\} \xrightarrow{\sigma} \{q_1,\dots,q_m\}$ in R such that:

- the node is labelled by σ:

$$\mathrm{lab}_\sigma(x);$$

- there are n edges incoming to x:

$$\mathrm{N}_{\mathrm{IN}}(e_1,x_1,\dots,e_n,x_n,x) : \mathrm{edge}^1(e_1,x_1,x) \wedge \dots \mathrm{edge}^1(e_n,x_n,x);$$

- each of those edges are distinct:

$$\textsc{Dist}(e_1,\dots,e_n) : \neg(e_1 = e_2) \wedge \dots \wedge \neg(e_{n-1} = e_n);$$

- there are no other edges incoming to x:

$$\textsc{Only}(e_1,\dots,e_n,x) : \exists e \exists y \mathrm{edge}^1(e,y,x) \Rightarrow (e = e_1) \vee \dots \vee (e = e_n);$$

- and the edges are in the set variables corresponding to $p_1,\dots,p_n$:

$$\textsc{States}(e_1,\dots,e_n,X_{p_1},\dots,X_{p_n}) : e_1 \in X_{p_1} \wedge \dots \wedge e_n \in X_{p_n}.$$

We do the same for the set of outgoing edges and the states $q_1,\dots,q_m$, defining the equivalent statement $\mathrm{n}_{\mathrm{out}}$ for n_{in}. We then combine all of these elements together to define an MSO statement that recognises the same DAGs as the automaton A:

$$\begin{aligned}
&\exists X_{q_1} \dots \exists X_{q_{|Q|}} \textsc{E-Part}(X_{q_1},\dots,X_{q_{|Q|}}) \wedge \forall x \textsc{Node}(x) \Rightarrow \\
&\bigvee_{\{p_1,\dots,p_n\} \xrightarrow{\sigma} \{q_1,\dots,q_m\} \in R} \Big(\mathrm{lab}_\sigma(x) \\
&\wedge \exists e_1 \exists x_1 \dots \exists e_n \exists x_n \big(\mathrm{N}_{\mathrm{IN}}(e_1,x_1,\dots,e_n,x_n) \wedge \textsc{Dist}(e_1,\dots,e_n) \\
&\wedge \textsc{Only}(e_1,\dots,e_n) \wedge \textsc{States}(e_1,\dots,e_n,X_{p_1},\dots,X_{p_n}) \big) \\
&\wedge \exists e'_1 \exists x'_1 \dots \exists e'_m \exists x'_m \big(\mathrm{M}_{\mathrm{OUT}}(e'_1,x'_1,\dots,e'_m,x'_m) \wedge \textsc{Dist}(e'_1,\dots,e'_m) \\
&\qquad \wedge \textsc{Only}(e'_1,\dots,e'_m) \wedge \textsc{States}(e'_1,\dots,e'_m,X_{q_1},\dots,X_{q_{|Q|}}) \big) \Big).
\end{aligned} \tag{4.23}$$

4.6 Probabilistic MSO

Recall that we are searching for a graph formalism that is both probabilistic and generates graph languages that are closed under intersection. It is clear that MSO languages are closed under intersection but it is not clear how to make them probabilistic.

Since the DAG automata languages are a subclass of the MSO languages, and in Chapter 3 we showed that there are DAG automata which cannot define probability distributions over their languages, this suggests that MSO may also not be probabilistic. However, this may not be the full story as it depends on how we use MSO to assign weights to a graph.

Droste and Gastin (2005) discusses weighted MSO on strings. They define the weights with respect to a semiring K. They essentially add an extra atomic formula to the MSO which corresponds to the weight and define rules for calculating the weights of formulas combined using the connectives and quantifiers of MSO. We could follow this method and say that in applying weights to the MSO statement for DAG automata, the weights can only appear in conjunction with the identification of which transition was applied at a node. In terms of the MSO formula for DAG automata above, this would mean adding a transition weight at the second line of Equation 4.23:

$$\bigvee_{\{p_1,\ldots,p_n\} \xrightarrow{\sigma} \{q_1,\ldots,q_m\} \in R} \mathrm{lab}_\sigma(x) \wedge w(\{p_1,\ldots,p_n\} \xrightarrow{\sigma} \{q_1,\ldots,q_m\}).$$

If we did this, then we could use the same argument as in Theorem 1 from Chapter 3 to show that MSO is not probabilistic. This is just one way of adding weights to the formula but it is not immediately clear that there is *no* way to add the weights in such a way that a probability distribution cannot be defined.

Reiter (2014) define the alternating distributed graph automaton as an automaton which recognises precisely the MSO-definable graph languages. To answer whether MSO on graphs can be probabilistic, it is likely worth studying these automata and considering how weights might be attached to them. We leave this as an open question.

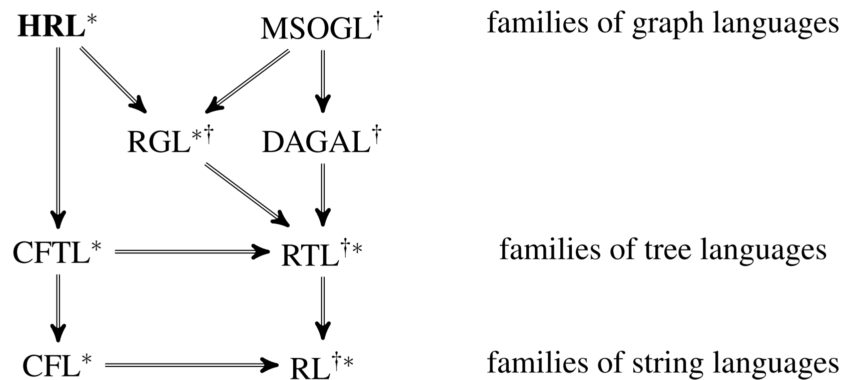
4.7 Conclusions

We study MSO on graphs due to the connection between the regular string and tree languages with the MSO string and tree languages, respectively. They share some desirable properties with these families, particularly that they are closed under intersection. However, it is not clear how to make MSO on graphs probabilistic and so we

do not wish to use MSO as our model over graphs. In the next chapter, we will look at the hyperedge replacement languages (HRL) which are opposite to MSO in terms of desirable properties—they are probabilistic but not closed under intersection. This leads us to look at subfamilies of the MSO graph languages and the HRL, particularly the regular graph languages. We show that they are a subfamily of both (using MSO transducers) and that they are both probabilistic and closed under intersection.

Chapter 5

Hyperedge replacement languages



In this chapter, we discuss the hyperedge replacement languages (HRL; Drewes et al. 1997). HRL are a generalisation of context-free string and tree languages to hypergraphs.[1] As we have mentioned throughout the thesis, we seek a probabilistic model of graphs that is closed under intersection. Similar to their string counterpart, HRL are not closed under intersection. This means that HRL do not possess the properties we desire. We will look at a restricted form of HRL in the next chapter—the regular graph languages (RGL; Courcelle 1991). To understand RGL, we first study HRL. We again repeat the Hasse diagram from previous chapters here, shown above.

5.1 Hyperedge replacement grammars

We repeat the formal definition of a hypergraph from Chapter 2 here.

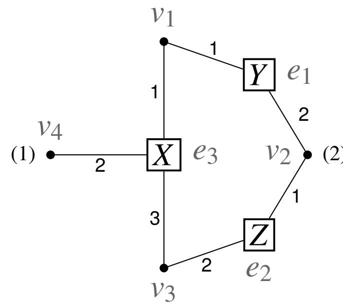
[1] Another generalisation to graphs is the node replacement grammars (Engelfriet, 1997). Song et al. (2017) use them in an NLP context for AMR-to-text generation. They are beyond the scope of this thesis

Definition 11. *A **hypergraph** over a ranked alphabet* Γ *is a tuple*

$$G = (V_G, E_G, att_G, lab_G, ext_G)$$

where V_G *is a finite set of nodes;* E_G *is a finite set of edges (distinct from* V_G*);* $att_G : E_G \to V_G^*$ *maps each edge to a sequence of nodes;* $lab_G : E_G \to \Gamma$ *maps each edge to a label such that* $|att_G(e)| = rank(lab_G(e))$*; and* ext_G *is an ordered subset of* V_G *called the **external nodes** of G.*

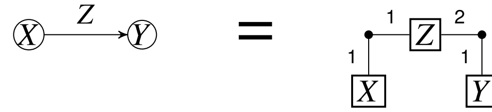
We assume that the elements of ext_G are pairwise distinct, and that the elements of $\text{att}_G(e)$ for each edge e are also pairwise distinct. An edge e is attached to its nodes by **tentacles**, each labelled by an integer indicating the node's position in $\text{att}_G(e) = (v_1, \ldots, v_k)$. The tentacle from e to v_i has label i, so the tentacle labels lie in the set $[k]$ where $k = \text{rank}(\text{lab}(e))$. To express that a node v is attached to the i-th tentacle of an edge e we say $\text{vert}(e,i) = v$. The nodes in ext_G are labelled by their position in ext_G. In figures, the i-th external node is labelled (i). We refer to nodes that are not external nodes as **internal nodes**, and they are unmarked in figures. The **rank** of an edge e is k if $\text{att}(e) = (v_1, \ldots, v_k)$ (or equivalently, rank(lab(e)) = k). The **rank** of a hypergraph G is $|\text{ext}_G|$.



Example 20. The hypergraph above has four nodes (shown as black dots) and three hyperedges labelled X, Y, and Z (shown boxed). The bracketed numbers (1) and (2) denote its external nodes and the numbers between edges and the nodes are tentacle labels. The grey labels next to the nodes and edges are variables so that we can refer to them in the text. Its definition would state:

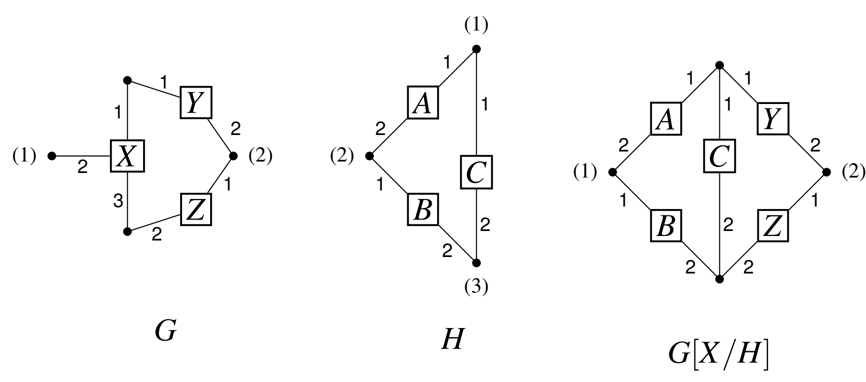
$$\begin{aligned}
\text{att}_G(e_1) &= (v_1, v_2) & \text{lab}_G(e_1) = Y \\
\text{att}_G(e_2) &= (v_2, v_3) & \text{lab}_G(e_2) = Z \\
\text{att}_G(e_3) &= (v_1, v_4, v_3) & \text{lab}_G(e_3) = X \\
ext_G &= (v_4, v_2).
\end{aligned}$$

Note that the hypergraphs as defined here are edge-labelled but not node-labelled. In Chapter 2, we showed how we can represent node labels in edge-labelled graphs by adding unary edges and labelling them by the node label. The figure below shows an example of this conversion for hypergraphs.



$HG_{\Sigma,\Gamma}$ is the set of all hypergraphs with node labels in Σ and edge labels in Γ. The hypergraphs we deal with here are in $HG_{\emptyset,\Gamma}$.

In generating a string using a context-free string grammar, the fundamental operation is replacing a single nonterminal X by a sequence of terminal and nonterminal symbols α if there is a production $X \to \alpha$ in the grammar. The corresponding operation in hyperedge replacement grammars is the replacement of a nonterminal edge by a hypergraph. Before we formally define replacement, we show an example.



Example 21. Replacement is shown in the figure above. We delete the X-labelled edge from G and plug in the graph H. We fuse a pair of nodes if: one is in G and was attached to the ith tentacle of X; and the other is the ith external node of H. In this case, three nodes are fused. We denote the replacement as $G[X/H]$ since the edge is unambiguous given its label.

To formally explain how to rewrite edges as graphs, we first need some notation. If f is a function and S is a set, $f|_S$ is the restriction of f to domain elements in S. If f,g are functions, $f \circ g$ is their composition.

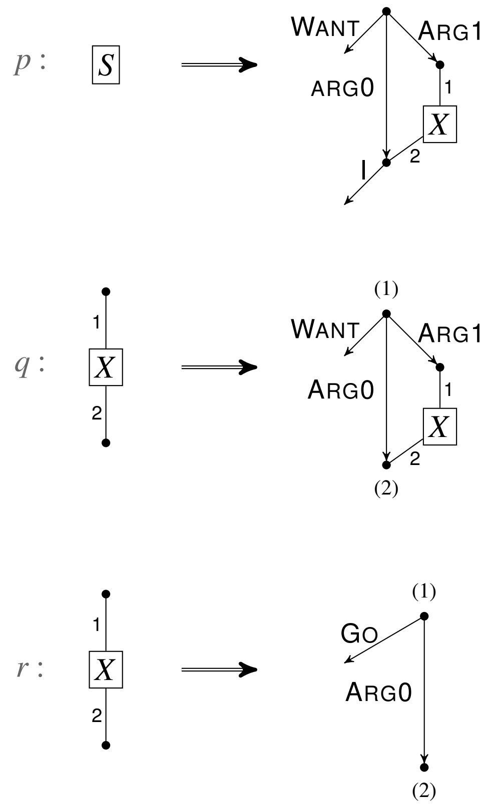
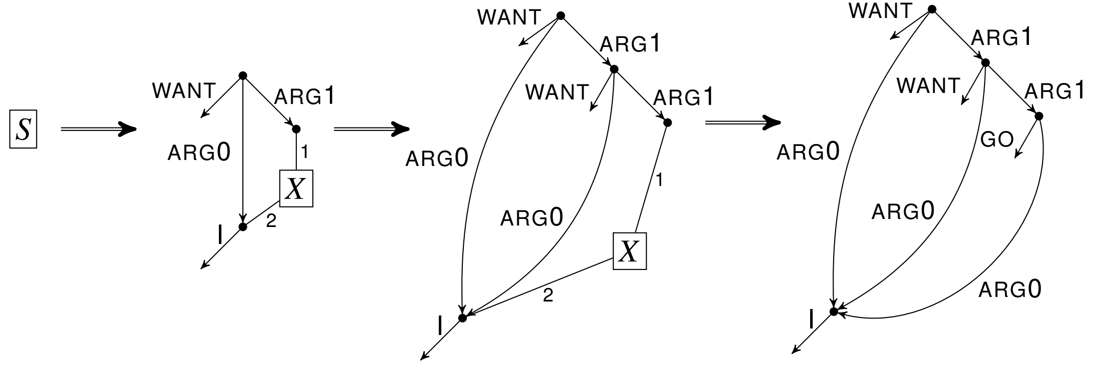


Figure 5.1: A HRG that generates a language of graphs representing sentences of the form "I want to want to ... want to go". The labels p,q,r are names for the productions so that we can refer to them in the text.

Definition 12. *Let G be a hypergraph with an edge e of rank k and let H be a hypergraph also of rank k disjoint from G. The **replacement** of e by H is the graph $G' = G[e/H]$. Let $V_{G'} = (V_G \cup V_H) - ext_H$, $E_{G'} = (E_G \cup E_H) - \{e\}$. Let $ext_H = (v_1, \ldots, v_k)$, $att_G(e) = (u_1, \ldots, u_k)$ and let $f : (V_G \cup V_H) \to V_{G'}$ replace v_i by u_i for $i \in [k]$ and be the identity otherwise. The extension of f to $(V_G \cup V_H)^*$ is also denoted f. Let $E = E_G - \{e\}$, then $att_{G'} = att_G|_E \cup (f \circ att_H)$, $lab_{G'} = lab_G|_E \cup lab_H$.*

Now that we have defined what it means to replace a nonterminal by a hypergraph, we can define a hyperedge replacement grammar.

Definition 13. *A **hyperedge replacement grammar** (HRG) $\mathcal{G} = (N_{\mathcal{G}}, T_{\mathcal{G}}, P_{\mathcal{G}}, S_{\mathcal{G}})$ consists of disjoint ranked alphabets $N_{\mathcal{G}}$ and $T_{\mathcal{G}}$ of nonterminals and terminals, a finite set of productions $P_{\mathcal{G}}$, and a start symbol $S_{\mathcal{G}} \in N_{\mathcal{G}}$. Every production in $P_{\mathcal{G}}$ is of the form $X \to H$ where $X \in N_{\mathcal{G}}$ is of rank k and H is a hypergraph of rank k over $N_{\mathcal{G}}$ and $T_{\mathcal{G}}$.*



Example 22. A hyperedge replacement grammar is shown in Figure 5.1. This grammar consists of three productions: p, q, and r. The nonterminals here are S and X (with S being the start symbol), and the terminals are WANT, I, GO, ARG0 and ARG1. This grammar generates graphs in the style of AMR (Banarescu et al., 2013). The figure above shows how this grammar can generate a graph representing the sentence "I want to want to go".

HRGs can generate languages of hypergraphs, and we will define precisely how below. For simplicity, in examples we show edges labelled by terminals of rank 1 or 2 only. We depict these edges as directed unary or binary edges with labels in small capitals. We depict edges labelled by nonterminals as boxes with the nonterminal written as a capital letter.

For each production $p : X \to G$, we use $\text{LHS}(p)$ to refer to its left-hand side (X) and $\text{RHS}(p)$ to refer to its right-hand side (G). An edge is a **terminal edge** if its label is a terminal and a **nonterminal edge** if its label is a nonterminal. A hypergraph is **terminal** if all of its edges are labelled with terminal symbols. An **induced subgraph** of a hypergraph G by edges $E' \subseteq E_G$ is the subgraph of G formed by including all edges in E' and their endpoints. The **terminal subgraph** of a hypergraph is the subgraph induced by its terminal edges.

Given a HRG $\mathcal{G}$, we say that hypergraph G **derives** hypergraph G', denoted $G \to G'$, iff there is an edge $e \in E_G$ and a nonterminal $X \in N_{\mathcal{G}}$ such that $\text{lab}_G(e) = X$ and $G' = G[e/H]$, where $X \to H$ is in $P_{\mathcal{G}}$. We extend the idea of derivation to its transitive closure $G \to^* G'$. For every $X \in N_{\mathcal{G}}$ we also use X to denote the connected hypergraph consisting of a single edge e with $\text{lab}(e) = X$ and nodes $(v_1, \ldots, v_{\text{rank}(X)})$ such that $\text{att}(e) = (v_1, \ldots, v_{\text{rank}(X)})$, and we define the language $\mathcal{L}_X(\mathcal{G}) = \{G \mid X \to^* G, G$ is terminal$\}$. The **language** of $\mathcal{G}$ is then $\mathcal{L}(\mathcal{G}) = \mathcal{L}_{S_{\mathcal{G}}}(\mathcal{G})$. We call the family of

languages that can be produced by any HRG the **hyperedge replacement languages (HRL)**.

5.2 HRGs are probabilistic but not intersectable

As depicted in the Hasse diagram at the beginning of the chapter, HRGs have probabilistic extensions. This holds due to the fact that we can assign weights to the productions of a HRG in the same way as we do with context-free string grammars and that we can always set these weights in a way such that the resulting weighted language defines a probability distribution.

Definition 14. *A **weighted** HRG is a HRG $\mathcal{G} = (N_\mathcal{G}, T_\mathcal{G}, P_\mathcal{G}, S_\mathcal{G})$ equipped with a weight function $w : P_\mathcal{G} \to \mathbb{R}$.*

Let G be a graph in $\mathcal{L}(\mathcal{G})$ and let $\mathbf{T}$ be a derivation tree of G. Then the weight of the tree $\mathbf{T}$ is

$$w(\mathbf{T}) = \prod_{t \in \mathbf{T}} w(\text{lab}(t)).$$

Let $\mathcal{T}(G)$ be the set of derivation trees of a hypergraph G. Then the weight of G is

$$w(G) = \sum_{\mathbf{T} \in \mathcal{T}(G)} w(\mathbf{T}).$$

Finally, the weight of a language L is

$$w(L) = \sum_{G \in L} w(G).$$

A weighted HRG is **probabilistic** if it defines a probability distribution over its language of hypergraphs (i.e. $w(G) \in [0,1]$ for each $G \in L$ and $w(L) = 1$). Unlike in the case of DAG automata, every HRG can define a a probability distribution (with full support) over its language.[2] Let $P_\mathcal{G}(X)$ be the set of productions with left-hand side X. One way that we can ensure that a weighted HRG is probabilistic is to set the weights such that $\sum_{p \in P_\mathcal{G}(X)} w(p) = 1$ for each nonterminal X (Booth and Thompson, 1973). This is a sufficient but not necessary condition for a HRG to be probabilistic.

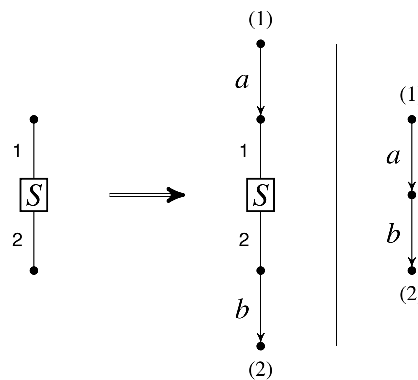
Besides probabilities, we have also expressed the desire for a formalism that produces languages that are closed under intersection. Unfortunately, this is not the case

[2] See Chapter 3 for the definition of a probability distribution with full support.

for hyperedge replacement languages. The proof is similar to that showing that the context-free string languages are not closed under intersection: it is decidable whether a HRG generates an empty language, but is not decidable whether the intersection of a pair of HRLs is empty—so the intersection may not be a HRL (Post, 1946).

5.3 Expressivity of HRGs

We can use HRGs to generate chain graphs (strings) by restricting the form of the productions in the grammars. They can generate the context-free string language $a^n b^n$. Compare $S \to aSb \mid ab$ with the grammar in the figure below.



They can actually generate string languages much more general than the context-free languages. Engelfriet and Heyker (1991) show that HRGs can simulate the class of mildly context-sensitive languages that is characterised, e.g. by linear context-free rewriting systems (LCFRS; Vijay-Shanker et al. 1987).

Example 23. Consider the mildly context-sensitive language $a^n b^n c^n$. A HRG can generate this language using the grammar in Figure 5.2: each production adds exactly one a, b, and c; and the as appear before all bs before all cs. Figure 5.3 shows how the grammar derives the string graph $a^3 b^3 c^3$. The arrows between the graphs show which production was used in that step and the red edges are those introduced at that step.

We can easily extend from this grammar to one that generates languages of the form $s_1^n s_2^n \dots s_m^n$ for strings $s_1, \dots, s_m$ and $m \in \mathbb{N}$. The maximum rank of a nonterminal in the grammar depends on how large m is: e.g. we need rank 2 nonterminals to generate $a^n b^n$; but we need rank 3 nonterminals to generate $a^n b^n c^n$.

The fact that HRGs has such a strong generative power when restricted to strings suggests that they are likely to be too powerful to model meaning representations. As

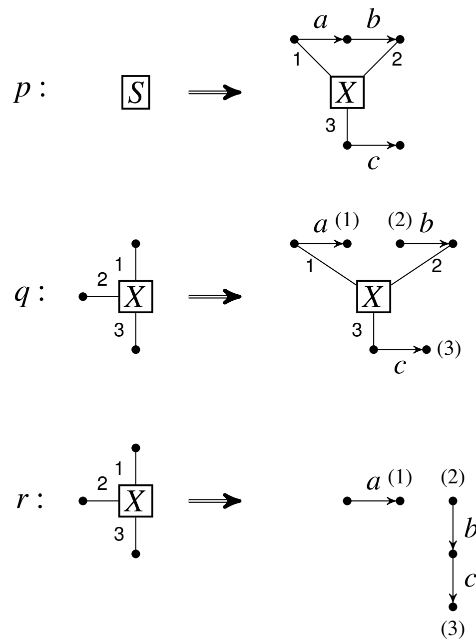


Figure 5.2: The HR grammar that generates the string language $a^n b^n c^n$.

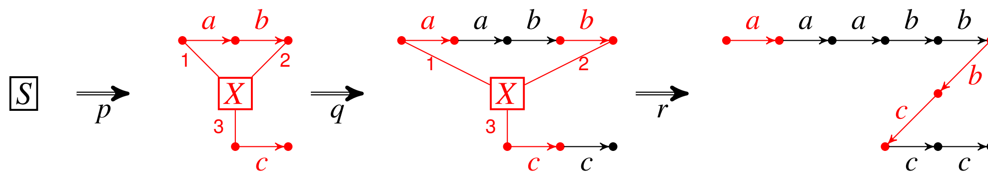


Figure 5.3: The derivation of the string $a^3b^3c^3$ using the HR grammar in Figure 5.2.

mentioned in Chapter 3, Drewes (2017) suggest that a desirable property of a formalism for modelling meaning representations would be to have regular path languages. Clearly, since HRG can generate context-free string languages and beyond, they do not have regular path languages.

5.4 HRLs are MSO transductions of RTLs

The set of derivation trees of a context-free string grammar is a regular tree language. The analogue is true for hyperedge replacement grammars, their derivation trees also form a regular tree language. The mapping from a derivation tree to a graph is an MSO transduction. We first define regular tree grammars.

Definition 15. *A regular tree grammar is a tuple* (N, Σ, P, S) *where:* N *is a finite set of*

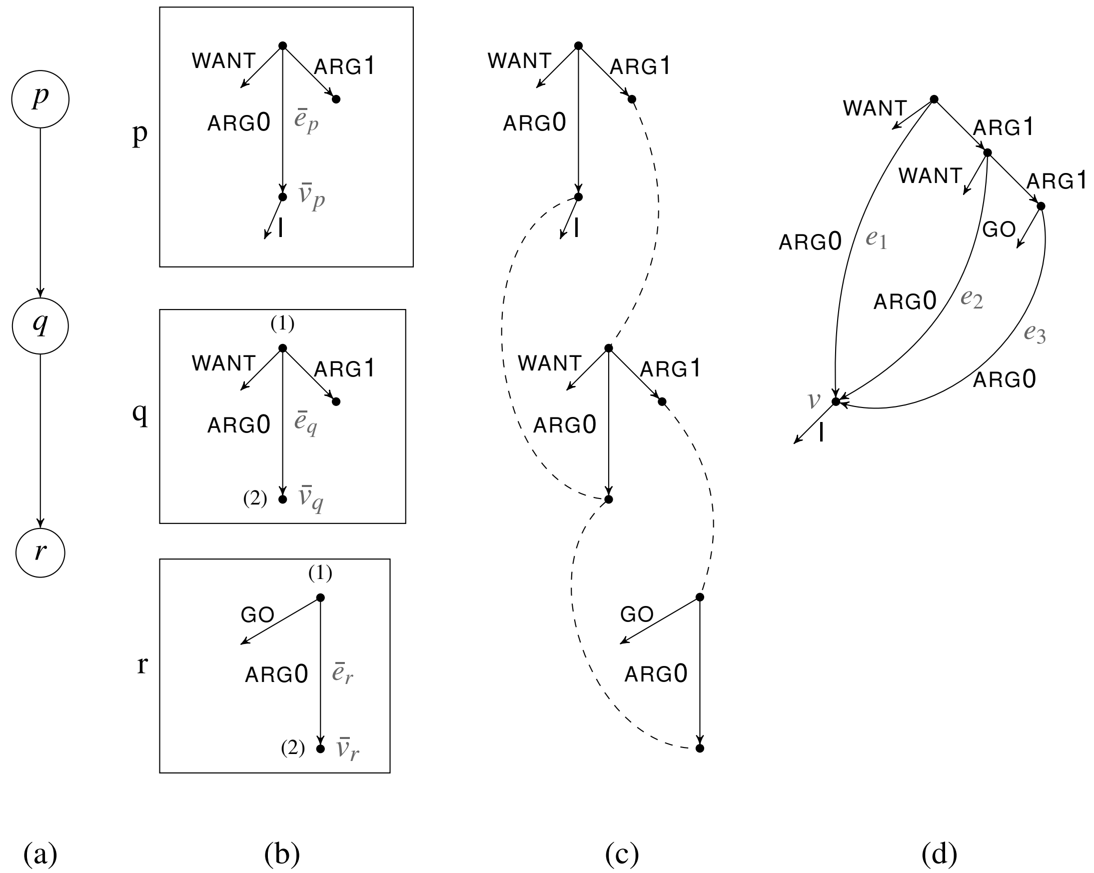


Figure 5.4: The derivation of the graph representing the sentence “I want to want to go” using the grammar shown in Figure 5.1 shown as a mapping from a derivation tree to a graph via an MSO transduction VAL. The labels in grey are for referencing specific nodes and edges in the text.

nonterminals; Σ *is a finite ranked alphabet (disjoint from* N*);* P *is a set of productions of the form* $X \to \sigma(X_1,\ldots,X_n)$ *for* $\sigma \in \Sigma$ *with rank* n *and* $X, X_1,\ldots,X_n \in N$*; and* $S \in N$ *is the start symbol.*

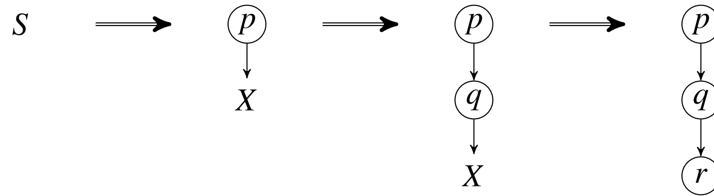
Example 24. Consider the regular tree grammar with nonterminals $\{S, X\}$, terminals $\{p,q,r\}$, productions

$$S \to p(X),$$

$$X \to q(X),$$

$$X \to r,$$

and S is the start symbol. The figure below shows the derivation of a tree using this grammar. The terminals are shown in circled nodes and the nonterminals as plain nodes.



For each HRG $\mathcal{G}$, let $\mathcal{T}_\mathcal{G}$ be the underlying regular tree grammar defining the derivation trees of the hypergraphs in $\mathcal{L}(\mathcal{G})$. Each $\mathbf{T} \in \mathcal{T}_\mathcal{G}$ has node labels in $P_\mathcal{G}$ and edge labels in $|\mathrm{NT}(P_\mathcal{G})|$. If a node has label p and $\textsc{rhs}(p)$ has n nonterminals $X_1,\ldots,X_n$ then for each $i \in [n]$, there is an i labelled edge from p to a node labelled q where $\textsc{lhs}(q) = X_i$. The label of the root of $\mathbf{T}$ must be p for some p with $\textsc{lhs}(p) = S$. Let $\textsc{val} : \mathcal{L}(\mathcal{T}_\mathcal{G}) \to \mathcal{L}(\mathcal{G})$ be a mapping from derivation trees to hypergraphs so that $G = \textsc{val}(\mathbf{T})$ iff $\mathbf{T}$ is a derivation tree of G. Since HRGs can be ambiguous, this mapping is not injective.

Courcelle (1991) shows that VAL is an MSO transduction.[3] The mapping VAL is the composition of two MSO transductions: the first takes each node in the derivation tree and creates a copy of the terminal subgraph of the production labelling that node; and the second decides which nodes need to be fused together to create the final graph. Courcelle (1991) shows that each mapping is an MSO transducer; their composition is also an MSO transducer.

[3]It can also be viewed in the related framework of interpreted regular tree grammars (Koller and Kuhlmann, 2011).

Example 25. Figure 5.4 shows the mapping from a derivation tree to a graph. The first mapping can be seen in the conversion from Figure 5.4(a) to Figure 5.4(b); and the second mapping can be seen in the conversion from Figure 5.4(c) to Figure 5.4(d). The underlying regular tree grammar for the HRG in Figure 5.1 is the one shown in Example 24.

This does not imply that HRL are MSO-definable since in general MSO-definability is not closed under MSO transductions (recall Example 19 from Chapter 4). Hence, an MSO transducer representing the inverse of VAL may not exist for an arbitrary HRG, but we later discuss a subfamily for which it does in Chapter 6, allowing us to apply the Backwards Translation Theorem (Theorem 3).

The transduction VAL preserves the terminal subgraph of every production used in a derivation and fuses nodes from different productions together in the output hypergraph. Node fusion is determined by an equivalence relation $\sim$ generated by a relation $\sim_0$. Let $\mathrm{NT}(p) = (e_1, \ldots, e_n)$ be the nonterminal edges of $\mathrm{RHS}(p)$, let $\mathrm{NT}_i(p) = e_i$, and let $\mathrm{ext}_G(i)$ be the ith external node of G.

To avoid confusion, we distinguish between elements of a hypergraph and its derivation tree. We denote a grammar by $\mathcal{G}$, hypergraph by G, derivation tree by $\mathbf{T}$, derivation tree node by $\mathbf{v}$, edges and nodes in productions are written with a bar ($\bar{v}$) and nodes and edges in G are unmarked (x). We will reuse this notation in the following two chapters.

Definition 16. *Let $\mathcal{G}$ be a HRG, $G \in L(\mathcal{G})$, and $\mathbf{T}$ be a derivation tree of $\mathcal{G}$, so that $G = \mathrm{VAL}(\mathbf{T})$. Define a binary relation $\sim_0$ on pairs $(\bar{x}, \mathbf{v})$ where $\bar{x}$ is a node in $\mathrm{RHS}(p)$ for some $p \in P_{\mathcal{G}}$ and $\mathbf{v}$ is a node of $\mathbf{T}$ with label p. Then $(\bar{x}, \mathbf{v}) \sim_0 (\bar{y}, \mathbf{v}')$ iff:*

1. *$\mathbf{v}, \mathbf{v}'$ are nodes in $\mathbf{T}$ and $\mathbf{v}'$ is the ith child of $\mathbf{v}$ in $\mathbf{T}$.*
2. *$p = lab_{\mathbf{T}}(\mathbf{v})$, $p' = lab_{\mathbf{T}}(\mathbf{v}')$.*
3. *$\bar{x}$ is the jth node of $NT_i(p)$, $\bar{y} = ext_{\mathrm{RHS}(p')}(j)$.*

We define $\sim$ as the reflexive, symmetric, transitive closure of $\sim_0$ (i.e. the equivalence relation generated by $\sim_0$). We previously described VAL as consisting of two mappings: one which generates subgraphs; and one which fuses nodes. The relation $\sim$ tells us which nodes to fuse in the second mapping.

Each node in a derived graph is the image of a node in a production along with a derivation tree node under a mapping h_v. Similarly, each edge in a graph is the image of a terminal edge in a production along with a derivation tree node under a mapping h_e. The mapping h_v is not injective since nodes can be fused. For example in Figure 5.4,

$\bar{v}_p$, $\bar{v}_q$, and $\bar{v}_r$ are all mapped to $\bar{v}$ in the graph. On the other hand, edges are never fused in the derivation of a graph and so the mapping h_e is injective. In the example, $\bar{e}_p$ is mapped to e_1, $\bar{e}_q$ is mapped to e_2, and $\bar{e}_r$ is mapped to e_3.

We now formally define this mapping. Let $E_P = \cup_{p \in P_G} E_{\text{RHS}(p)}$ and $V_P = \cup_{p \in P_G} V_{\text{RHS}(p)}$. Then $h_e : E_P \times V_{\mathbf{T}} \to E_G$ maps a pair $(\bar{e}, \mathbf{v})$ to its image e in the hypergraph, where $\bar{e}$ is a terminal edge in p and $\text{lab}(\mathbf{v}) = p$. $h_v : V_P \times V_{\mathbf{T}} \to V_G$ maps a pair $(\bar{x}, \mathbf{v})$ to its image v, where $\bar{x}$ is a node in p and $\text{lab}(\mathbf{v}) = p$.

5.5 Formal properties of HRGs

In the next chapter, we will discuss a restricted form of HRGs called the regular graph grammars (RGG). We will step through a proof that shows that the languages generated by RGG are MSO-definable. We will require several lemmas to establish this result. In this chapter, we provide two results that hold for HRG in general that we will use in the next chapter.

Before we state the first lemma, we consider the example in Figure 5.4 to get an intuition for it. Take the nodes $\bar{v}_p$ and $\bar{v}_r$ in (b). We know that both of those nodes get mapped to the same node in the graph, v in (d). We can also see that in (a) that p is the grandparent of r in the derivation tree. The node between p and r is the node q. The lemma will establish that there must be some node in the production q that also maps to v in the graph, in this case that is $\bar{v}_q$ shown in (b).

Lemma 1. *If $(\bar{x}, \mathbf{v}) \sim (\bar{x}', \mathbf{v}')$ then for the unique path in $\mathbf{T}$ connecting $\mathbf{v}$ and $\mathbf{v}'$ consisting of nodes $\mathbf{v}_1, \ldots, \mathbf{v}_k$, for each $i \in [k]$ there exists a unique node $\bar{x}_i$ in $\text{RHS}(lab_{\mathbf{T}}(\mathbf{v}_i))$ such that $(\bar{x}_i, \mathbf{v}_i) \sim (\bar{x}, \mathbf{v})$.*

Proof. We prove by induction on the length of the path.

Base Case:

There is a single node $\mathbf{v}_1$ between $\mathbf{v}$ and $\mathbf{v}'$. By the construction of $\sim$, $(\bar{x}, \mathbf{v}) \sim (\bar{x}', \mathbf{v}')$ is constructed from relations $(\bar{x}, \mathbf{v}) \sim (\bar{x}_1, \mathbf{v}_1)$ and $(\bar{x}_1, \mathbf{v}_1) \sim (\bar{x}', \mathbf{v}')$ for some $\bar{x}_1 \in \text{RHS}(p_1)$ where $p_1 = \text{lab}_{\mathbf{T}}(\mathbf{v}_1)$. There can only be one such node $\bar{x}_1 \in \text{RHS}(p_1)$ since two nodes from the same production can never be fused; this comes from the fact that we assume that both the external nodes of a graph are pairwise distinct, and the nodes of an edge for each edge are also pairwise distinct.

Assumption:

If there are fewer than n nodes between $\mathbf{v}$ and $\mathbf{v}'$ in $\mathbf{T}$ then each node $\mathbf{v}_i$ on the path has a corresponding unique node $\bar{x}_i$ in $\textsc{rhs}(\mathrm{lab}_{\mathbf{T}}(\mathbf{v}_i))$ such that $(\bar{x}_i, \mathbf{v}_i) \sim (\bar{x}, \mathbf{v})$.

Inductive Case:

Let $\mathbf{v}$ and $\mathbf{v}'$ be such that there are n nodes separating them in $\mathbf{T}$ and let $(\bar{x}, \mathbf{v}) \sim (\bar{x}', \mathbf{v}')$. Then since $\mathbf{v}$ and $\mathbf{v}'$ are not connected via an edge of $\mathbf{T}$, by the construction of $\sim$, there must be some $\mathbf{v}''$ on the path between $\mathbf{v}$ and $\mathbf{v}'$ such that $(\bar{x}, \mathbf{v}) \sim (\bar{x}'', \mathbf{v}'')$ and $(\bar{x}'', \mathbf{v}'') \sim (\bar{x}', \mathbf{v}')$. The length of the path from $\mathbf{v}$ to $\mathbf{v}''$ is less than n and so for each $\mathbf{v}_i$ between $\mathbf{v}$ and $\mathbf{v}''$ there exists a unique $\bar{x}_i$ such that $(\bar{x}, \mathbf{v}) \sim (\bar{x}_i, \mathbf{v}_i)$. Similarly, this holds for all nodes on the path between $\mathbf{v}''$ and $\mathbf{v}'$.

Therefore, for each node $\mathbf{v}_i$ on the path from $\mathbf{v}$ to $\mathbf{v}'$ there is a unique $\bar{x}_i$ such that $(\bar{x}_i, \mathbf{v}_i) \sim (\bar{x}, \mathbf{v})$. □

Before we state the second lemma, we again refer back to the example in Figure 5.4. Again take the node v in the graph in (d). This node is the image of both $\bar{v}_p$ and $\bar{v}_r$ from (b). We can see from the figure that $\bar{v}_p$ is an internal node in p, and that $\bar{v}_r$ is an external node in r. The following lemma will establish that because $\bar{v}_p$ is internal, then $\bar{v}_r$ must be external and also that this implies that p is an ancestor of r in the derivation tree—which can be seen in (a).

Lemma 2. *Let $\mathcal{G}$ be a HRG, and let G be a hypergraph in $\mathcal{L}(\mathcal{G})$ with derivation tree* $\mathbf{T}$. *If $\bar{x}$ and $\bar{x}'$ are nodes such that $h_v(\bar{x}, \mathbf{v}) = h_v(\bar{x}', \mathbf{v}')$ with $\mathbf{v} \neq \mathbf{v}'$ and, if $\bar{x}$ is internal in* $\textsc{rhs}(p)$ *for $p = lab_{\mathbf{T}}(\mathbf{v})$, then $\bar{x}'$ is an external node of* $\textsc{rhs}(p')$ *for $p' = lab_{\mathbf{T}}(\mathbf{v}')$ and* $\mathbf{v}$ *is an ancestor of* $\mathbf{v}'$ *in* $\mathbf{T}$.

Proof. We prove by induction on the length of the path between $\mathbf{v}$ and $\mathbf{v}'$.

Base Case:

There is a path of length 1 between $\mathbf{v}$ and $\mathbf{v}'$. Then there is an edge connecting $\mathbf{v}$ and $\mathbf{v}'$. Since $h_v(\bar{x}, \mathbf{v}) = h_v(\bar{x}', \mathbf{v}')$, it holds that $(\bar{x}, \mathbf{v}) \sim (\bar{x}', \mathbf{v}')$. Since there is a single edge between $\mathbf{v}$ and $\mathbf{v}'$, either $(\bar{x}, \mathbf{v}) \sim_0 (\bar{x}', \mathbf{v}')$ holds or $(\bar{x}', \mathbf{v}') \sim_0 (\bar{x}, \mathbf{v})$ holds. By the definition of $\sim_0$, for $(\bar{x}', \mathbf{v}') \sim_0 (\bar{x}, \mathbf{v})$ to hold, $\bar{x}$ must be external. However, here $\bar{x}$ is internal and so $(\bar{x}, \mathbf{v}) \sim_0 (\bar{x}', \mathbf{v}')$ must hold. This implies that $\bar{x}'$ is external in $\textsc{rhs}(p')$, and $\mathbf{v}$ is the parent (and therefore ancestor) of $\mathbf{v}'$ in $\mathbf{T}$.

Assumption:

Assume that if the path is of length less than n then $\bar{x}'$ is an external node of $\textsc{rhs}(p')$ and $\mathbf{v}$ is an ancestor of $\mathbf{v}'$ in $\mathbf{T}$.

Inductive Case: Let the path between $\mathbf{v}$ and $\mathbf{v}'$ be of length n. Let $\mathbf{v}_{n-1}$ be the node on the path that shares an edge with $\mathbf{v}'$. Then by Lemma 1 there is some $\bar{x}_{n-1} \in$

$\text{RHS}(\text{lab}_{\mathbf{T}}(\mathbf{v}_{n-1}))$ such that $(\bar{x},\mathbf{v}) \sim (\bar{x}_{n-1},\mathbf{v}_{n-1})$ and $(\bar{x}_{n-1},\mathbf{v}_{n-1}) \sim (\bar{x}',\mathbf{v}')$. By the inductive assumption, $\bar{x}_{n-1}$ is external in $\text{RHS}(\text{lab}_{\mathbf{T}}(\mathbf{v}_{n-1})$ and $\mathbf{v}$ is an ancestor of $\mathbf{v}_{n-1}$. Since $\mathbf{v}_{n-1}$ and $\mathbf{v}$ share an edge, either $(\bar{x}_{n-1},\mathbf{v}_{n-1}) \sim_0 (\bar{x}',\mathbf{v}')$ or $(\bar{x}',\mathbf{v}') \sim_0 (\bar{x}_{n-1},\mathbf{v}_{n-1})$ hold.

Case 1: $(\bar{x}_{n-1},\mathbf{v}_{n-1}) \sim_0 (\bar{x}',\mathbf{v}')$. $\bar{x}'$ is external by the definition of $\sim_0$ and $\mathbf{v}_{n-1}$ is a parent of $\mathbf{v}'$. Since $\mathbf{v}$ is an ancestor of $\mathbf{v}_{n-1}$, $\mathbf{v}$ is also an ancestor of $\mathbf{v}'$.

Case 2: $\mathbf{v}_{n-1}$ is a child of $\mathbf{v}'$. Since $\mathbf{v}$ is an ancestor of $\mathbf{v}_{n-1}$, this means that $\mathbf{v}'$ then lies along the path between $\mathbf{v}$ and $\mathbf{v}_{n-1}$. The length of the path from $\mathbf{v}$ to $\mathbf{v}_{n-1}$ is $n-1$ and so this implies that the length of the path from $\mathbf{v}$ to $\mathbf{v}'$ is $n-2$, which is a contradiction.

Therefore, $\bar{x}'$ must be external and $\mathbf{v}$ is an ancestor of $\mathbf{v}'$. □

As a consequence of this, we reach the following remark which we will use in Chapter 6.

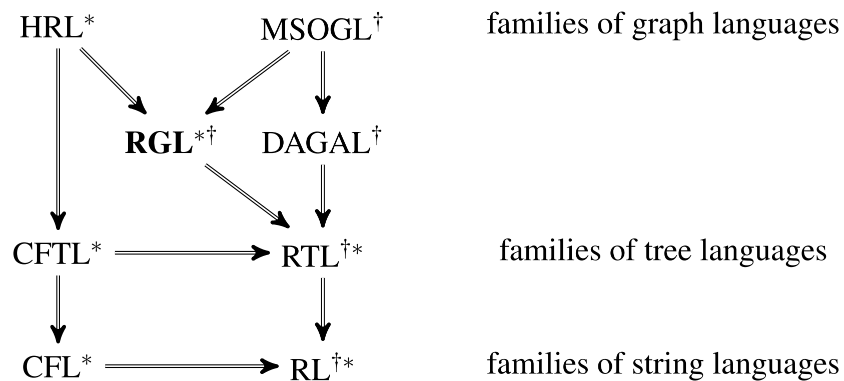
Remark 1. *If $h_v(\bar{x},\mathbf{v}) = h_v(\bar{x}',\mathbf{v}')$ then $\bar{x}$ and $\bar{x}'$ cannot both be internal.*

5.6 Conclusions

HRGs are a context-free rewriting system for generating languages of graphs. They are a generalisation of the context-free string and tree languages. By adding weights to the productions in HRGs, we can use them to define probabilistic languages. However, they are not closed under intersection and so are not suitable to be used as a “finite-state” model over graphs which we seek to define. They are also very expressive and may be too powerful to be useful in modelling meaning representations (Drewes, 2017). In Chapter 6, we will look at the regular graph languages (RGL; Courcelle 1991), a subfamily of HRL. Building on the Lemmas 1 and 2, we will show that RGL are also a subfamily of the MSO graph languages. We will also discuss two other subfamilies of HRL—the restricted DAG languages (Björklund et al., 2016), and the tree-like languages (Matheja et al., 2015).

Chapter 6

(Re)introducing regular graph languages



At the outset of this thesis, we expressed a desire for a graph formalism that is both probabilistic and intersectable. So far, we have looked at the DAG automata languages, the hyperedge replacement languages, and the monadic second-order graph languages. All these formalisms either have no known probabilistic extension or are not closed under intersection. This leads us to consider the families of graph languages that are subfamilies of both HRL and MSOGL, with the hope that they inherit the desirable properties of both.

Courcelle (1991) define the family of languages comprising the intersection of the HR and MSO graph languages as the **strongly context-free** languages (SCFL). The definition is non-constructive—there is no grammar formalism defining this family. However, the same paper goes on to define regular graph grammars (RGG) as a subfamily of SCFL.

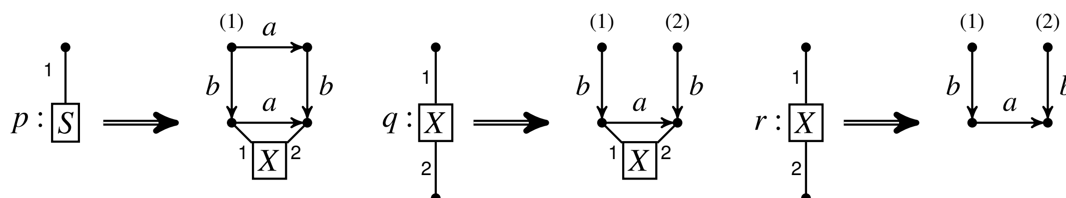
An RGG is a restricted form of hyperedge replacement grammar (see Chapter 5). It was invented as a means to prove a conjecture in Courcelle (1991) that if a graph language is both recognisable and can be generated by a HRG, then it is MSO-definable.[1] As far as we know, they have not been widely studied since their first publication. Particularly, they have not been studied in the context of NLP. We seek to use them as a possible candidate formalism that is both probabilistic and closed under intersection.

The focus of this chapter will be to show that RGL is a subfamily of both HRL and MSOGL. Since RGG is defined as a restricted form of HRG, it is trivial to show that RGL is a subfamily of HRL. A sketch of this proof can be found in Courcelle (1991). Here, we frame the proof for an NLP audience, where the aim is to make it more accessible. We also provide many more details and formal proofs that were not in the original paper. Courcelle (1991) mention without proof that RGL are closed under intersection. We prove that they are on Page 108.

The proof that RGL is a subfamily of MSOGL relies on the backwards translation theorem—for an MSO transducer τ and a language L: if L is MSO-definable then so is $\tau^{-1}(L)$. Recall that, like their string counterpart, the set of derivation trees of a HRL is a regular tree language (and therefore is MSO-definable). This means that if we can construct an MSO transducer τ which maps graphs in an RGL L_G to their derivation trees L_T and $\tau^{-1}(L_T) = L_G$ then it holds that L_G is MSO-definable. In this chapter, we construct this transducer and prove that for a given RGL, it outputs its derivation trees. The construction depends on the restricted form of the productions in RGG and does not work for arbitrary HRG.

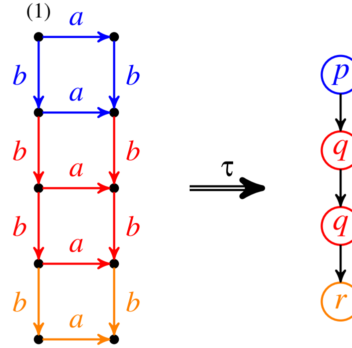
6.1 Informal description of RGG

Before we formally define RGG, we show an example of an RGG and its output under the MSO transducer from graphs to derivation trees. The grammar shown below is an RGG, and we name its productions p, q and r so that we can refer to them.



[1]This conjecture was recently proved by Bojańczyk and Pilipczuk (2016).

The MSO transducer first partitions the edges of the graph, each matching the right-hand side of a production. Then for each of these subgraphs, it outputs a derivation tree node labelled by the relevant production. In the figure below, the blue edges correspond to production p, the red edges to production q, and the orange edges to production r. Edges are drawn between the derivation tree nodes in the output if their subgraphs connect together in the right way in the graph.



Not all HRLs are MSO-definable. Therefore, given a HRG, there is not always an MSO transducer that takes the language of graphs of the HRG and outputs its set of derivation trees. We will show how the restrictions of RGGs make it possible to construct such an MSO transducer for them. One of the main intuitions is that each time a production is applied, we add a set of connected edges to the graph—e.g. above, all the blue edges are connected. This property is useful for MSO since it can perform local "checks", like it does for FSAs in §4.1, on the graph to see if there is a connected subgraph that matches the right-hand side of a production.

6.2 Formal definition of RGG

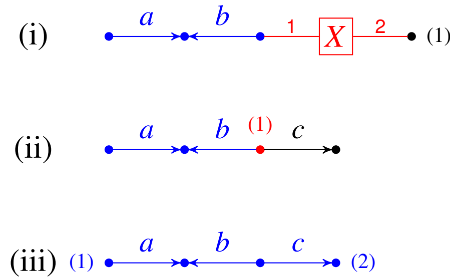
Before we show that RGLs are MSO-definable, we first study their definition, along with some examples of languages they can and cannot generate. We begin with some definitions necessary to define RGL.

Definition 17. *Given a hypergraph G, a **path** in G from a node v to a node v' is a sequence*

$$(v_0, i_1, e_1, j_1, v_1)(v_1, i_2, e_2, j_2, v_2)\dots(v_{k-1}, i_k, e_k, j_k, v_k)$$

such that $vert(e_r, i_r) = v_{r-1}$ and $vert(e_r, j_r) = v_r$ for each $r \in [k]$, $v_0 = v$, and $v_k = v'$. The length of this path is k.

Part of the definition of RGG depends on the types of paths appearing in the right-hand sides of productions. In particular, it requires terminal and internal paths. Recall from Chapter 5 that an internal node is any node that is not an external node. A path is **terminal** if every edge in the path is labelled by a terminal. A path is **internal** if each v_i is internal for $1 \leq i \leq k-1$. The endpoints v_0 and v_k of an internal path can be external.



Example 26. The paths in the figure above illustrate what it means to be terminal and internal. Each path starts from the left and is marked in blue until it either reaches the end or is blocked by a nonterminal edge or an external node (shown in red). Note that a path here does not have to follow the direction of the edges. Path (i) is an internal path since the only external node occurs as an endpoint, however it is not terminal since to reach the right-most node, we must pass through the nonterminal edge labelled X. Path (ii) is terminal since all edges are labelled by terminals but it is not internal since the second node from the right is an external node. Finally, path (iii) is terminal since all the edge labels are terminal and it is also internal since only the endpoints are external.

Definition 18. *A HRG $\mathcal{G} = (N_\mathcal{G}, T_\mathcal{G}, P_\mathcal{G}, S_\mathcal{G})$ is a* ***regular graph grammar*** *if each nonterminal in $N_\mathcal{G}$ has rank at least one and for each $p \in P_\mathcal{G}$ the following hold:*

(C1) $\textsc{rhs}(p)$ *has at least one edge.*

(C2) If $\textsc{rhs}(p)$ *has any internal nodes, then every edge in* $\textsc{rhs}(p)$ *must be connected to at least one internal node; if* $\textsc{rhs}(p)$ *has no internal nodes, then it must be a single terminal edge connecting all nodes.*

(C3) Every pair of nodes in $\textsc{rhs}(p)$ *is connected by a terminal and internal path.*

As defined in Chapter 5, $\textsc{rhs}(p)$ refers to the graph on the right-hand side of production p. From now on, we will use $\mathcal{G} = (N, T, P, S)$ as it is usually clear which grammar we are referring to. In any place where there might be confusion, we will use subscripts.

Example 27. Table 6.1 shows a HRG that is also an RGG. We will use this as a running example throughout the chapter.

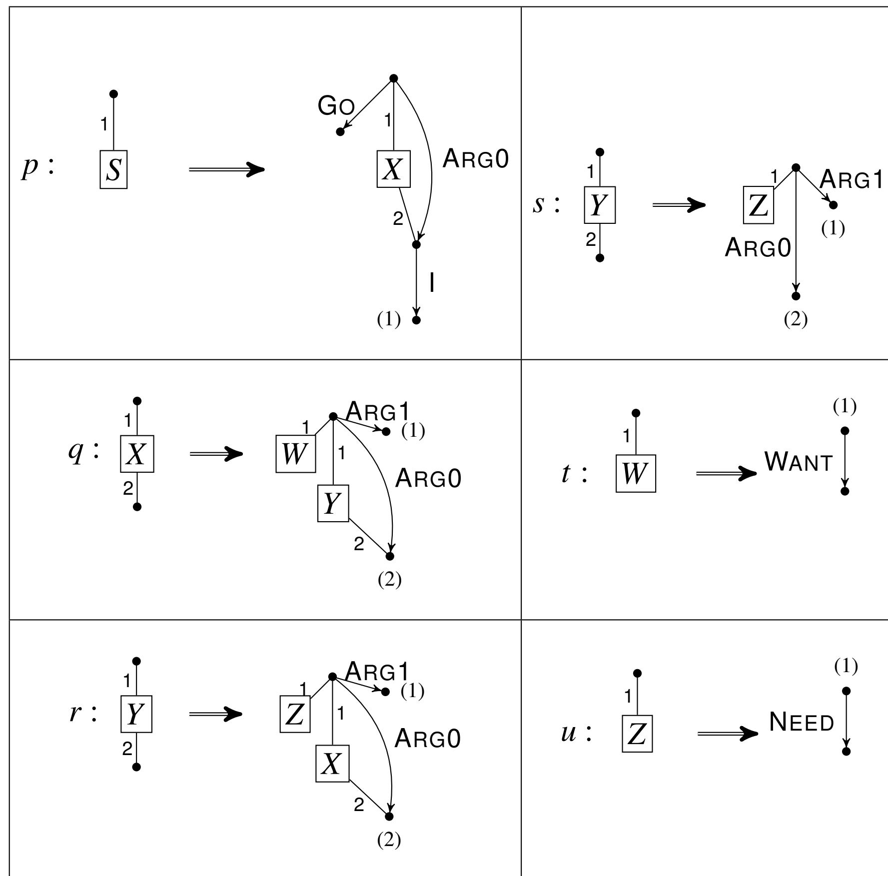


Table 6.1: An RGG. Each production satisfies the requirements for a HRG to be an RGG as in Definition 18. The labels p, q, r, s, t, and u label the productions so that we can refer to them in the text. Note that Y can be rewritten either via production r or s.

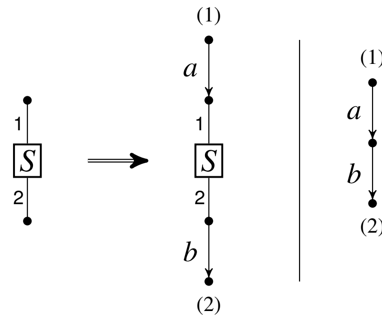
Remark 2. *Note that the dangling edges in Table 6.1 are no longer unary as they were in Chapter 5. This is due to the restrictions of RGG. If we were to write the edges as unary then in the p production, either the top node or the node at the bottom of the* ARG0 *edge would have to be external. Then either the* GO *edge or the* I *edge would have no internal node.*

We will address this remark again at the end of the chapter when discussing the limitations of RGG.

6.3 Expressivity of RGG

RGG is less expressive than HRG. For example, the HRG generating the string language $a^n b^n$ is not an RGG. The figure below shows this grammar. It is not an RGG because there is no terminal path between the two external nodes in the first production,

any path has to pass through the nonterminal S.



Not only is the above grammar not an RGG, but no HRG generating $a^n b^n$ is an RGG. We expand on this in the following lemma. In Chapter 5, we discussed HRG generating string languages, we now consider the string languages generated by RGG.

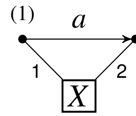
Proposition 1. *The string languages that RGG can generate are the regular string languages.*

Proof. Consider the form of a right-hand side production of an RGG generating languages of strings. It is clear that the terminal edges in any right-hand side need to form a string structure since they are what will appear in the derived graph. Since this is an RGG, any terminal edges must be connected to one another (via the terminal and internal path property). This means that the terminal subgraph of the right-hand side is a string.

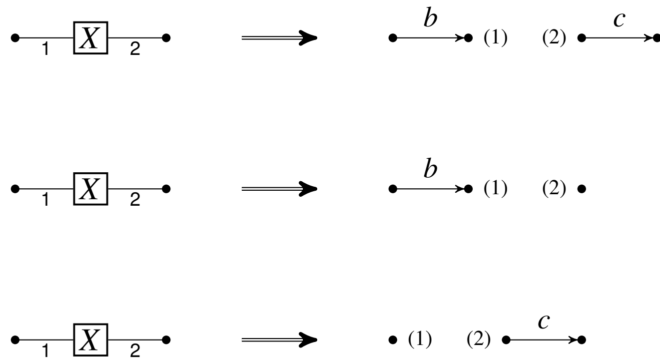
We now deal with the nonterminal edges and which nodes they are attached to in this terminal string. Nonterminal edges can only be attached to endpoints of the terminal string. If we allowed nonterminals to be attached to nodes internal to the terminal string then when they rewrite, there would have to be a terminal generated at that point (due to the fact that the definition of RGG implies that there will always be a terminal edge attached to each external node). Therefore, nonterminals may only be attached at endpoints of the string and may not be attached to any other nodes—if they were then there would be nodes only attached to nonterminal edges, which is not allowed in RGG.

Since we assume that the elements of $\mathrm{att}_G(e)$ are pairwise distinct for each $e \in G$, we only need to deal with nonterminals or rank 1 or 2. We begin by looking at nonterminals of rank 2. A nonterminal of rank 2 means that it must be attached to both ends of the terminal string like in the figure below. This production must have one internal node and one external node. There must be at least one external node as

every nonterminal (and therefore every production) has rank at least one, and there can only be one external node as otherwise the edges not attached to any internal nodes. Without loss of generality, we choose the left node to be external.



Consider how the nonterminal X above would rewrite. To terminate the derivation and generate a string, it would have to rewrite in a form of one of the following ways:



None of these productions is allowed in RGG. If the production was not terminating then a nonterminal would be attached somehow to the nodes. However, the terminal and internal path condition would still be violated and so would not be allowed in RGG.

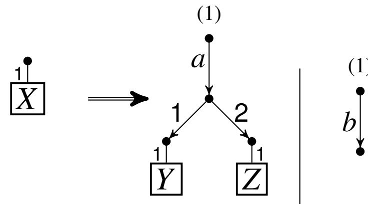
Therefore, the nonterminals must have rank no more than 1 and appear attached to the endpoints of terminal strings. Since in RGG, all nonterminals have rank at least one, the nonterminals must have rank exactly one and the productions must be of the form:

Productions with the start symbol on the left-hand-side are the only ones allowed to take the form of the top production, the rest must take the form of the bottom production. This grammar is in the form of a right-linear context-free string grammar, which generates the regular string languages.[2]

[2]If in the top production, we put Y on the left of a then we could have a mirrored version of the lower production. This would simulate a left-linear context-free string grammar.

□

Similarly, when restricted to trees, RGGs can produce only the regular tree languages. The figure below shows a tree-generating RGG that generates binary trees the internal nodes of which are represented by a-labelled edges, and the leaves of which are represented by b-labelled edges.[3] This is in the same format as a regular tree grammar, which only allows states or nonterminals to appear as leaves. Regular tree grammars generate ordered trees, this order is simulated using edge labels (e.g. the edges labelled 1 and 2 in the example).



Now we consider the *graph* languages that RGG can generate. As mentioned at the beginning of the chapter, RGGs do not define SCFL. To show this, we take the BEACHBALL language (shown in Figure 6.1) which can be generated by a HRG and is MSO-definable.

A HRG generating the BEACHBALL language is shown in Figure 6.2, meaning that it lies in HRL. The MS_2 statement defining the BEACHBALL language is defined over alphabet $\{a\}$ by saying that: (1) there are exactly two nodes in the graph:

$$\exists x_1 \exists x_2 \text{NODE}(x_1) \wedge \text{NODE}(x_2) \wedge \neg(x_1 = x_2) \wedge \forall x_3 (\text{NODE}(x_3) \Rightarrow (x_3 = x_1) \vee (x_3 = x_2));$$

and (2) every edge in the graph is labelled a and goes from the first node to the second node:

$$\forall e \text{EDGESETS}(e) \Rightarrow (\text{lab}_a(e) \wedge \text{edge}^2(e, x_1, x_2)).$$

Therefore, the BEACHBALL language is MSO-definable.

Proposition 2. *RGG cannot generate the* BEACHBALL *language.*

[3]The format of the trees generated by the RGG is different to that of a regular tree grammar which would have node labelled terminals. If we were to write the node labels as unary edges dangling from the root node (as we did in Chapter 5), then the productions would not satisfy (C2) of the definition of RGG. However, there is a one-to-one mapping between the trees generated by the RGG and those generated by a regular tree grammar.

Proof. In the BEACHBALL language, graphs can have an unbounded number of edges. Since productions contain a fixed number of edges in their right-hand sides, this means that there are graphs in the BEACHBALL language which require an unbounded number of production applications.

However, each graph in this language contains exactly two nodes. For an RGG to generate a language with an unbounded number of productions, it must apply productions with nonterminals an unbounded number of times. By (C2) of the definition of RGG, productions containing a nonterminal must have at least one internal node. Each time such a production is applied, a new node appears in the graph. Therefore, an RGG cannot generate the BEACHBALL language. □

The above proposition implies that RGL is a *proper* subclass of SCFL.

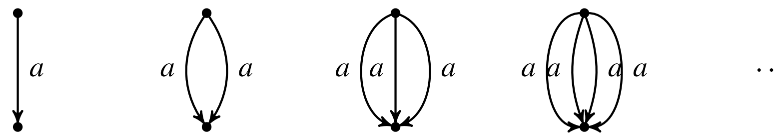


Figure 6.1: The BEACHBALL language: every graph must have exactly two nodes, and must have one or more a-edges from one to the other (always in the same direction). This language cannot be generated by an RGG.

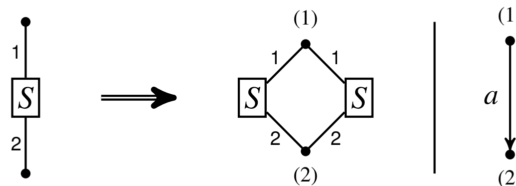


Figure 6.2: The HRG which can generate the BEACHBALL language. This is not an RGG.

6.4 RGL is a subfamily of HRL and MSOGL

We begin to set up the proof now that RGL is a subfamily of HRL and MSOGL.

Example 28. Figure 6.3 shows the transformation VAL of a derivation tree into a graph using the grammar shown in Table 6.1. This is the same transformation as we saw in Figure 5.4.

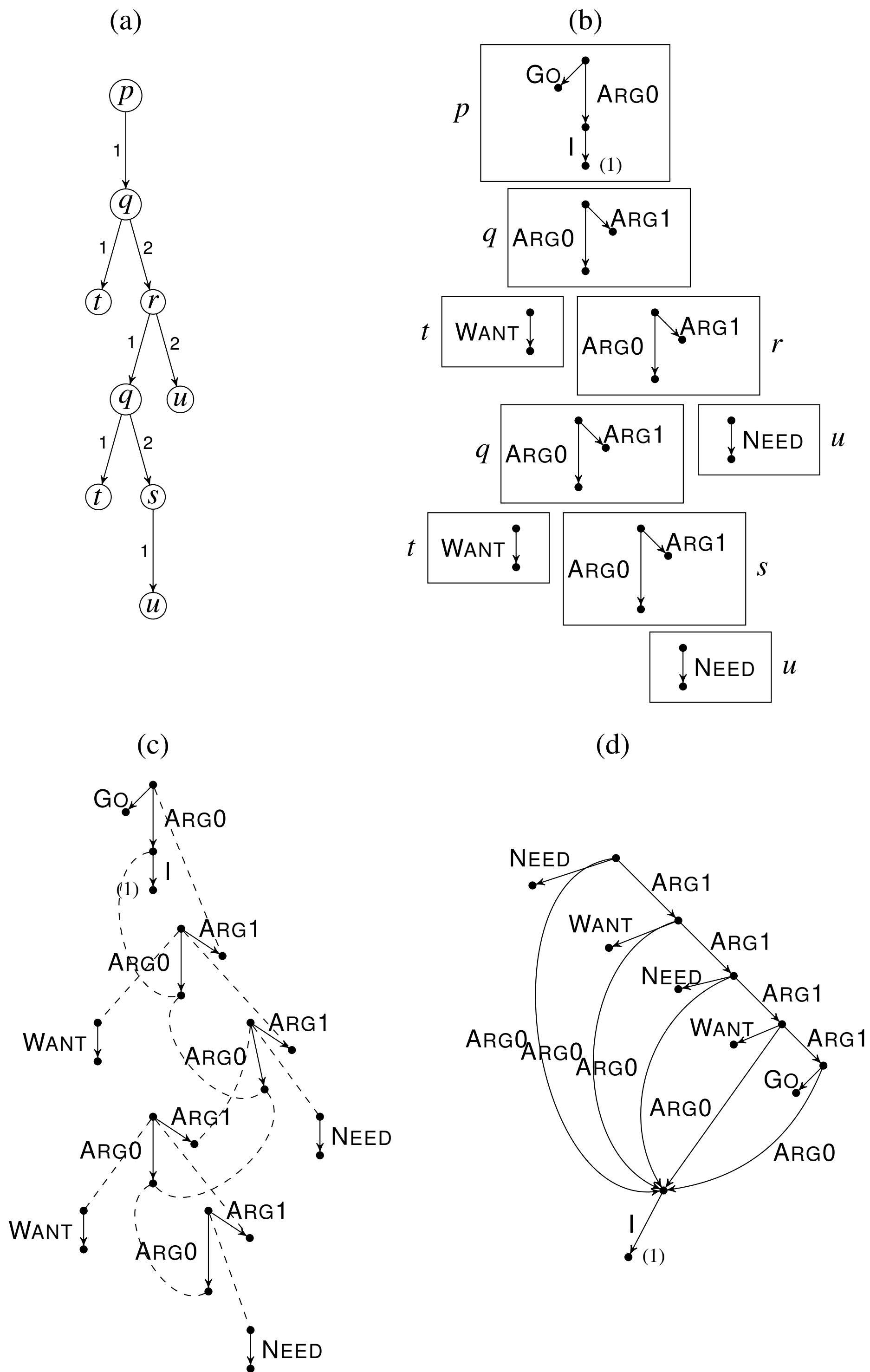


Figure 6.3: Using the RGG shown in Table 6.1 we show here: a derivation tree (a), the terminal subgraphs of every copy of every production in the derivation tree (b), the relation $\sim$ illustrated with dashed lines (c) and the resulting graph (d).

RGLs are HRLs by definition; we will prove that they are also MSO graph languages by constructing the inverse of VAL (described in Chapter 5), a transducer from RGL graphs to their derivation trees. Since the derivation trees are MSO-definable, RGLs must also be MSO-definable by Theorem 3 which we repeat below.

Theorem 3 (Backwards Translation Theorem)**.** *If L is an MSO-definable graph language and f is an MSO graph transduction then $f^{-1}(L) = \{G \mid f(G) \in L\}$ is MSO-definable.*

In the context of Figure 6.3, we essentially want to reverse this transformation and get from the complete graph, shown in (d), to the derivation tree, shown in (a). We define some concepts to make this idea concrete.

Each of the nodes and edges in the graph in Figure 6.3(d) exist because they are the image of some nodes or edges in the productions. The **pre-image** of an edge in a derived graph is the unique edge in a production that it is the image of. Although a node in a derived graph may be the image of many production nodes, it can be the image of at most one internal node (see Lemma 2). Therefore, we define the pre-image of a node to be the internal node it is the image of.

The construction requires a unique **anchor** element for each production in the grammar. If the production contains internal nodes, then the anchor can be any internal node; if there are no internal nodes, then by definition there is a single terminal edge and this is chosen to be the anchor. Given an input graph, the transducer first guesses—via parameter assignment—the pre-image of each edge and the set of elements whose pre-images are anchors. It then checks that the graph can be partitioned into a set of connected subgraphs—each of which is isomorphic to the terminal subgraph of some production.

If these constraints are satisfied, the transducer outputs a node corresponding to each guessed anchor and an edge between anchors that it identifies to be in a parent-child relationship. The output nodes are labelled by the productions they were anchors for. This then forms a derivation tree for the input graph.

Every valid parameter assignment corresponds to a different output from the transducer. We will show that for each input graph in the language, all its derivation trees lie in this output set. We will also show that for any input graph outside of the language, there are no derivation trees lying in the output set.

Theorem 4. *RGL $\subseteq$ MSO graph languages.*

The proof of Theorem 4 is provided in §6.5.2.

6.4.1 Anchors and parameters

This section sets up the parameters and precondition of the transducer. The parameters try to guess which nodes and edges in the grammar correspond to the nodes and edges in the input graph. The precondition performs checks on this parameter assignment to see if it is allowed under the grammar.

First, we define the parameters. There are two types of productions in RGGs: those with a single terminal edge, all nodes of which are external; and those where each edge has an internal node. We call the former **ext-productions** and the latter **int-productions**. For each int-production, we arbitrarily choose one of its internal nodes to be its anchor. For each ext-production, we choose its single terminal edge to be the anchor. By Lemma 2, this choice ensures that a pair of anchors cannot be fused, so the set of anchors in any derived graph is guaranteed to be in one-to-one correspondence with the nodes of its derivation tree. Note that this is not in general true for HRGs, since they do not require all productions to have some internal node.

We define two sets of parameters: $\mathcal{E}$ and $\mathcal{C}$, where $\mathcal{E}$ guesses pre-images of edges, and $\mathcal{C}$ guesses anchors (which may be either nodes or edges).

To define $\mathcal{E}$ precisely, we require some notation. Recall from the previous chapter that we denote a grammar by $\mathcal{G}$, hypergraph by G, derivation tree by $\mathbf{T}$, derivation tree node by $\mathbf{v}$, edges and nodes in productions are written with a bar ($\bar{v}$) and nodes and edges in G are unmarked (x).

Let $\mathcal{G} = (N,T,P,S)$ be an RGG, and for each $p \in P$, let $\mathrm{T}(p) = \{\bar{f}_{p,1},\ldots,\bar{f}_{p,|\mathrm{T}(p)|}\}$ enumerate the terminal edges of $\textsc{rhs}(p)$ and let $\gamma_{p,j}$ be the label of $\bar{f}_{p,j}$ for each $p \in P$ and $j \in [|\mathrm{T}(p)|]$. Let $\mathrm{NT}(p) = \{e_1,\ldots,e_n\}$ enumerate the nonterminal edges in $\textsc{rhs}(p)$, let $|\mathrm{NT}(p)|$ be the number of nonterminal edges in p, and let $|\mathrm{NT}(P)| = \max_{p\in P}|\mathrm{NT}(p)|$. Given a node $\mathbf{v}$ in a derivation tree $\mathbf{T}$, we say that $\mathbf{v}$ is an i-child if it is the i-th child of some other node in $\mathbf{T}$. By convention, the root node is the only 0-child. For example, in Figure 6.3, the top-most q node is a 1-child and the right-most u node is a 2-child.

Let G be in $\mathcal{L}(\mathcal{G})$ and let $\mathbf{T}$ be a derivation tree of G. For each $i \in [0,|\mathrm{NT}(P)|]$, $p \in P$ and $j \in [|T(p)|]$, we define a parameter $E_{i,p,j}$:

$$E_{i,p,j} = \{e \in E_G \mid e = h_e(\bar{f}_{p,j},\mathbf{v}) \text{ and } \mathbf{v} \text{ is an } i\text{-child.}\}$$

Let $\mathcal{E} = \{E_{i,p,j}\}$ for $i \in [0,|\mathrm{NT}(P)|]$, $p \in P$ and $j \in [|\mathrm{T}(p)|]$.

Define

$$\mathcal{C} = \{u \in V_G \mid u = h(\bar{c}_p,\mathbf{v}), \mathrm{lab}_{\mathbf{T}}(\mathbf{v}) = p\}$$

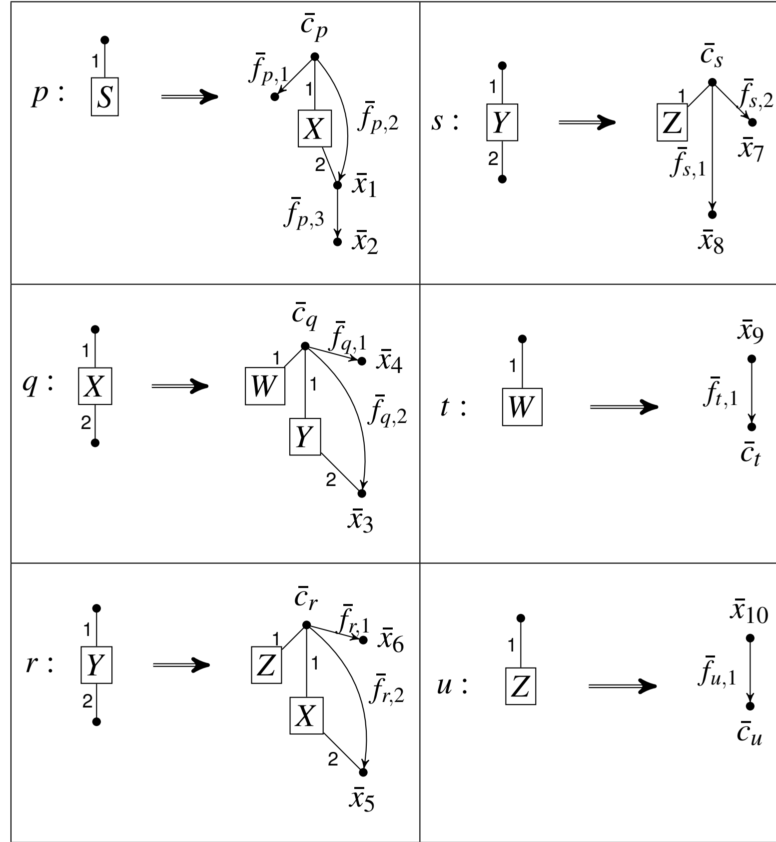


Table 6.2: The productions from Table 6.1 with variable names added to each of the nodes and terminal edges. Node variables of the form $\bar{c}_x$ for $x \in \{p,q,r,s,t,u\}$ indicate anchors.

where $h = h_e \cup h_v$ since $\bar{c}_p$ can either be an edge or a node.

Let $\mathcal{W} = \mathcal{E} \cup \mathcal{C}$.

Example 29. Table 6.2 shows the productions of Table 6.1 with labels on each node and edge. Figure 6.4 shows the derivation tree and graph from Figure 6.3 with variable names added. We use these variable names to refer to specific nodes and edges in the text. For example, $h_v(\bar{c}_s, \mathbf{v}_8) = v_1$, and $h_e(\bar{f}_{u,1}, \mathbf{v}_9) = e_5$.

Example 30. Using the labels in Table 6.2 and Figure 6.4, we see that $E_{0,p,1} = \{e_9\}$, $E_{1,q,2} = \{e_{11}, e_{13}\}$, and $v_1 = h(\bar{c}_s, \mathbf{v}_8)$ is an anchor.

6.4.2 Path properties of RGLs

In the high-level description of the proof at the beginning of the chapter, we mentioned that the MSO transducer checks that a subgraph is isomorphic to the terminal subgraph of a production in the grammar. A natural question is: how does the MSO check this?

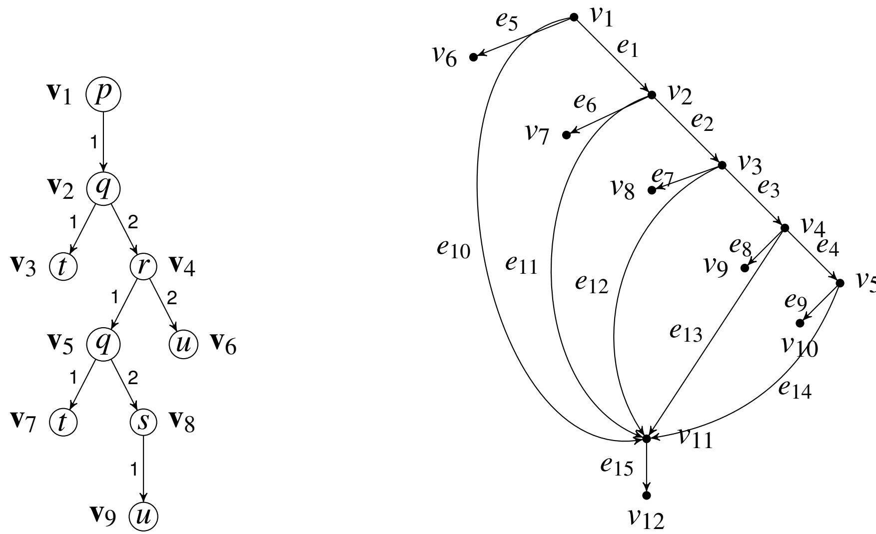


Figure 6.4: The derivation tree from Figure 6.3(a) and the graph from Figure 6.3(d) with variable names for the nodes and edges for referencing in the text. We will repeat sections of this figure throughout the examples, the variable names will stay the same as in this.

It does so by looking at the paths from the anchors to every other node in a production and identifying the corresponding paths in the graph. By (C3) of the definition of RGG, there is a terminal and internal path from the anchor to every other node in the production—this section shows how the projections of those paths can be identified in the graph.

Let $\mathcal{G}$ be an RGG, $G \in \mathcal{L}(\mathcal{G})$, and let $\mathbf{T}$ be a derivation tree of G. In the following, we relate paths within individual productions in P (denoted π) to paths in G (denoted λ). For each e in G, we define $\mathrm{o}(e) = (i, p, j)$ iff $e \in E_{i,p,j}$.

For every path λ in G of the form

$$(v, i_1, e_1, j_1, v_1)(v_1, i_2, e_2, j_2, v_2)\dots(v_{k-1}, i_k, e_k, j_k, v')$$

we define its **trace** as the sequence

$$\mathrm{tr}(\lambda) := (\mathrm{o}(e_1), i_1, j_1)(\mathrm{o}(e_2), i_2, j_2)\dots(\mathrm{o}(e_k), i_k, j_k).$$

Now let π be a path

$$(\bar{v}, i_1, \bar{e}_1, j_1, \bar{v}_1)\dots(\bar{v}_{k-1}, i_k, \bar{e}_k, j_k, \bar{v}')$$

in $\mathrm{RHS}(p)$ for some $p \in P$. Let $\mathbf{v} \in V_{\mathbf{T}}$, $p = lab_{\mathbf{T}}(\mathbf{v})$. We denote by $h(\pi, \mathbf{v})$ the following

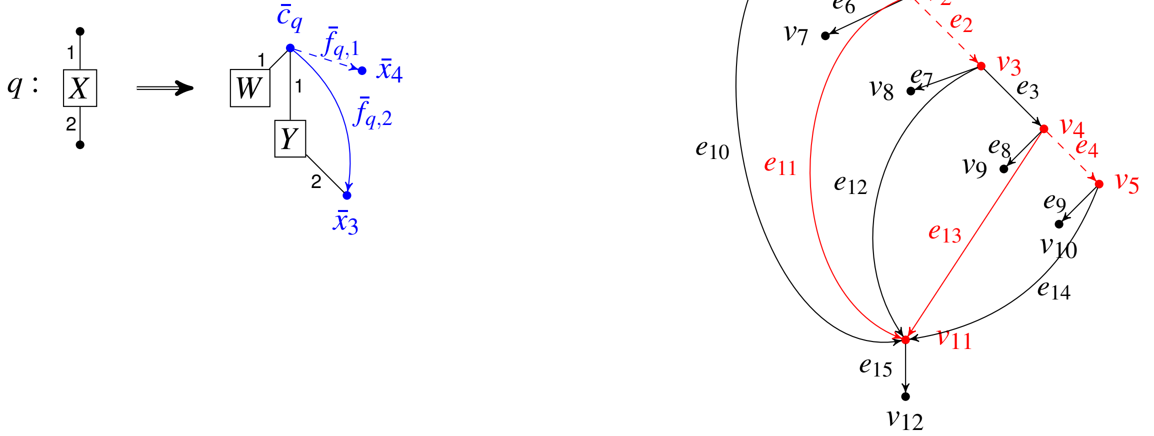
path in G:

$$(h(\bar{v},\mathbf{v}),i_1,h(\bar{e}_1,\mathbf{v}),j_1,h(\bar{v}_1,\mathbf{v}))\ldots(h(\bar{v}_{k-1},\mathbf{v}),i_1,h(\bar{e}_k,\mathbf{v}),j_1,h(\bar{v}',\mathbf{v}))$$

If $\mathbf{v}$ is an i-child of some node in $V_{\mathbf{T}}$ then $\mathrm{tr}(h(\pi,\mathbf{v}))$ is the sequence

$$((i,p,m_1),i_1,j_1)\ldots((i,p,m_k),i_k,j_k)$$

where $\bar{e}_j = \bar{f}_{p,m_j}$ for each $j \in [k]$. Note that $\mathrm{tr}(\pi) = \mathrm{tr}(h(\pi,\mathbf{v}))$. The trace is a property that remains constant when a path is projected from a production into a graph. This projection is not one-to-one since a production can be applied several times; a trace appears in the graph once for each application of the corresponding production in a derivation. For $\mathbf{v} \in V_{\mathbf{T}}$, we write $\pi \in \mathrm{RHS}(\mathrm{lab}_{\mathbf{T}}(\mathbf{v}))$ to denote that π is a path in the production which is the label of $\mathbf{v}$.



Example 31. Let π be the path $(\bar{x}_3,2,\bar{f}_{q,2},1,\bar{c}_q)(\bar{c}_q,1,\bar{f}_{q,1},2,\bar{x}_4)$ in production q in the figure above shown in blue. The two images of this path in the graph are highlighted in red in the graph. The left path is $h(\pi,\mathbf{v}_5) = (v_{11},2,e_{11},1,v_2)(v_2,1,e_2,2,v_3)$ and the right path is $h(\pi,\mathbf{v}_2) = (v_{11},2,e_{13},1,v_4)(v_4,1,e_4,2,v_5)$. The trace of both of these paths is the same and is $((1,q,2),2,1)((1,q,1),1,2)$.

This example shows that since a production can be applied many times, the trace of a path in a production does not uniquely define a path in the derived graph. Consider the path $(\bar{c}_q,1,\bar{f}_{q,1},2,\bar{x}_4)$ which is the second half of the path we have been looking at, shown dashed above. This path also has two images in the graph, again both shown

dashed and in red. The trace of both of these paths is $((1,q,1),1,2)$. However, there is no other path starting from either v_2 or v_4 with this trace. The following lemma establishes that for certain forms of path in a production, we can fully specify the image of the path given just the trace and the starting node in the graph.

Lemma 3 (Lemma 5.5 from Courcelle (1991))**.** *Let $\mathcal{G}$ be an RGG, G be a graph in $\mathcal{L}(\mathcal{G})$, and $\mathbf{T}$ be a derivation tree of G. Let λ be a path in G of the form $h(\pi,\mathbf{v})$ for some $\mathbf{v} \in V_{\mathbf{T}}$ and some terminal path $\pi \in \textsc{rhs}(lab_{\mathbf{T}}(\mathbf{v}))$. The final node of π may be internal or external but every other node must be internal. If λ' is another path in G with the same trace and the same initial node as λ, then $\lambda' = \lambda$.*

Proof. Let

$$\lambda = (v,e_1,i_1,j_1,v_1)\ldots(v_{k-1},e_k,i_k,j_k,v_k)$$

and

$$\lambda' = (v,e'_1,i'_1,j'_1,v'_1)\ldots(v'_{k'-1},e'_{k'},i'_{k'},j'_{k'},v'_{k'})$$

be the two paths. Their traces are the same and so $(i_\eta, j_\eta, \mathrm{o}(e_\eta)) = (i'_\eta, j'_\eta, \mathrm{o}(e'_\eta))$ for each $\eta \in [k]$ and $\eta' \in [k']$. This implies that $k = k'$, and $i_\eta = i'_\eta$, $j_\eta = j'_\eta$ for all $\eta \in [k]$. The initial node of both paths is the same and so $v = h_v(\bar{v},\mathbf{v}) = h_v(\bar{v}',\mathbf{v}')$ for some $\bar{v}$ in $\textsc{rhs}(p)$ such that $\mathrm{lab}_{\mathbf{T}}(\mathbf{v}) = p$, and $\mathbf{v}'$ in $\mathbf{T}$ such that $\bar{v}'$ in $\textsc{rhs}(p')$ and $\mathrm{lab}_{\mathbf{T}}(\mathbf{v}') = p'$. Let $e_1 = h_e(\bar{e}_1,\mathbf{v})$, and $e'_1 = h_e(\bar{e}'_1,\mathbf{v}')$ for some $\bar{e}_1$ in $\textsc{rhs}(p)$ and $\bar{e}'_1$ in $\textsc{rhs}(p')$.

Base Case: We shall prove that $\bar{e}_1 = \bar{e}'_1$ and $\mathbf{v} = \mathbf{v}'$. Assume that $\mathbf{v} \neq \mathbf{v}'$. We know that $v = h_v(\bar{v},\mathbf{v}) = h_v(\bar{v}',\mathbf{v}')$. $\bar{v}$ is internal since all but the last node of π must be internal. Therefore, Lemma 2 implies that $\bar{v}'$ must be external in $\textsc{rhs}(p')$. λ and λ' have the same trace and so if $\mathrm{o}(e_1) = (i,p,j)$ then $\mathrm{o}(e'_1) = (i,p,j)$ then $p = p'$ and so $\bar{v}'$ is external in $\textsc{rhs}(p)$. From the trace, it also follows that $\bar{e}_1 = \bar{e}'_1$ since they are both the j-th terminal edge of $\textsc{rhs}(p)$. Hence, $\bar{v}$ and $\bar{v}'$ are both the i_1^{st} node of $\bar{e}_1$ in $\textsc{rhs}(p)$ and so since $\bar{v}'$ is an external node then $\bar{v}$ is also an external node. We have reached a contradiction since $\bar{v}$ is an internal node. Therefore, $\mathbf{v} = \mathbf{v}'$. The fact that $\mathbf{v} = \mathbf{v}'$ and $\bar{e}_1 = \bar{e}'_1$ implies that $e_1 = e'_1$. Using the trace, this also shows that $\bar{v}_1 = \bar{v}'_1$ since they are both the j_1^{st} node of $\bar{e}_1$ in $\textsc{rhs}(p)$ and so $v_1 = v'_1$.

Assumption: Assume that $v_\eta = v'_\eta$ and $e_\eta = e'_\eta$ for all $\eta < n \leq k$.

Inductive Case: Consider the n-th node of λ, v_n, and the n-th node of λ', v'_n. By the inductive hypothesis, $v_{n-1} = v'_{n-1}$. The node $\bar{v}_{n-1}$ such that $v_{n-1} = h_v(\bar{v}_{n-1},\mathbf{v})$ is internal in $\textsc{rhs}(p)$ because $\bar{v}_{n-1}$ is not the final node in π. We are now back in the same

position as the beginning of the base case and we can repeat the argument as before by letting $v = v_{n-1}$ and $v_1 = v_n$. Therefore, $v_n = v'_n$ and $e_n = e'_n$ for all $\eta \in [k]$ and so $\lambda' = \lambda$. □

Lemma 3 implies that for any terminal internal path in a production which starts from an internal node; given the image of the initial node in the graph and the trace of the path; we can uniquely determine the image of the entire path in the graph. In terms of the example showing the paths in red and blue above, the full path starts from an external node and we can see that there are two paths starting from v_{11} with the same trace. The shorter (dashed) path starts from an internal node and this time there is one path starting from each of v_2 and v_4 with the same trace.

For int-productions, there is such a path from the anchor to each other node (by property C3 of RGGs). We will use this fact in Lemma 5 to show that we can uniquely relate an anchor to the rest of the nodes generated by the same production application. In the case of ext-productions, all paths from an anchor to a node are of the form $\pi = (\bar{e}, i, \bar{v}_j)$, where e is the single terminal edge; the image of such a path is also uniquely determined in the graph as they can be described as the j-th node of the anchor edge.

6.4.3 MSO for hypergraphs

In Chapter 4, we defined MSO over graphs. Here, we define MSO over hypergraphs since RGL (and HRL) are families of hypergraph languages. We use MS_2 in this chapter when dealing with hypergraphs and define a hyperedge e of rank n as:

$$\text{edge}^2(e, x_1, \ldots, x_n),$$

with label:

$$\text{lab}_a(e)$$

where a also has rank n. In Chapter 4, we defined the formula $\textsc{Inc}(e,x)$ to represent that a node x is attached to an edge e. Here we use $\textsc{Inc}_i(e,x)$ to represent that x is the i-th node of e which is defined as follows:

$$\textsc{Inc}_i(e,x) : \exists x_1 \ldots \exists x_{i-1} \exists x_{i+1} \ldots \exists x_n \left(\text{edge}^2(e, x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)\right).$$

In the case that we do not need to specify which tentacle a node is attached to, we simply use $\textsc{Inc}(e,x)$ as before. We also use the formula $\textsc{Node}(x)$ from Chapter 4 to represent that x is a node.

6.4.4 The precondition of the transducer

We define the transducer in terms of an RGG $\mathcal{G}$ as a mapping from graphs in $\mathcal{L}(\mathcal{G})$ to their derivation trees. The first part of the transducer is the precondition ρ. The precondition is an MSO formula which the input must satisfy. We will show that for each $G \in \mathcal{L}(\mathcal{G})$, $G \models \rho$. It is important to note here though that $\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\rho)$ as there may be other graphs satisfying the precondition that are not in $\mathcal{L}(\mathcal{G})$. However, when we define the output formulas for the transducer, we will show that for any graph G, $\tau(G)$ contains a derivation tree of $\mathcal{L}(\mathcal{G})$ if and only if $G \in \mathcal{L}(\mathcal{G})$. We will use this fact in conjunction with the backwards translation theorem to show that $\mathcal{L}(\mathcal{G})$ is MSO-definable. This is in contrast to Courcelle (1991), which sketches a precondition which is satisfied by a graph if and only if the graph is in the language. We believe that our presentation is simpler and more accessible.

The precondition is a conjunction of two formulas which state that:

- the assignment of edges to $\mathcal{E}$ is valid for a graph in $\mathcal{L}(\mathcal{G})$,
- and the graph can be decomposed into a set of connected subgraphs, each corresponding to the application of a production.

We first define some concepts that will be used in the construction of the precondition. If X is a nonterminal in N, then $P_X = \{p \in P \mid \textsc{lhs}(p) = X\}$, and an X-derivation tree is a derivation tree with respect to X as the start symbol (in this case, the root will have label in P_X). An S-derivation tree is referred to simply as a derivation tree.

Let $G \in \mathcal{L}_X(\mathcal{G})$ and $q : X \to H$ such that H has nonterminals $Y_1, \ldots, Y_n$ and $G = H[Y_1/H_1]\ldots[Y_n/H_n]$. Then $H_\eta \in \mathcal{L}_{Y_\eta}(\mathcal{G})$ for each $\eta \in [n]$. Let $\mathbf{T}_\eta$ be a derivation tree for H_η and let $\alpha_{\mathbf{T}_\eta}$ be the assignment of $\mathcal{W}$ to the nodes and edges in H_η. Then we can define $\alpha_{\mathbf{T}}(\mathcal{E})$ in terms of the set of $\alpha_{\mathbf{T}_\eta}(\mathcal{E})$s:

$$\alpha_{\mathbf{T}} : \begin{cases} e \in E_{i,p,j} & \text{if } e \in E_{H_\eta}, \alpha_{\mathbf{T}_\eta} : e \in E_{i,p,j}, i \neq 0 \\ e \in E_{\eta,p,j} & \text{if } e \in E_{H_\eta}, \alpha_{\mathbf{T}_\eta} : e \in E_{0,p,j} \\ e \in E_{0,q,j} & \text{if } e \in E_H, e = h_e(\bar{f}_{q,j}, \mathbf{v}_0). \end{cases} \tag{6.1}$$

Where $e = h_e(\bar{f}_{q,j}, \mathbf{v}_0)$ means that e can be uniquely identified as corresponding to $\bar{f}_{q,j}$ since H and $\textsc{rhs}(q)$ are isomorphic and $\mathbf{v}_0$ is the root of $\mathbf{T}$. For the anchor set,

$$\alpha_{\mathbf{T}}(\mathcal{C}) = c \cup_{\eta \in [n]} \alpha_{\mathbf{T}_\eta}(\mathcal{C}) \tag{6.2}$$

where $c = h(\bar{c}_q, \mathbf{v}_0)$.

6.4.4.1 Edge requirements

The edge requirements consist of three main parts which we call E1, E2 and E3. E1 ensures that for each edge, we choose exactly one edge in the grammar that it can be the image of. E2 checks that the choice made by E1 obeys the label of the edges since we cannot change the label of an edge from it being in the grammar to a derived graph. E3 then identifies the unique root of the derivation of the graph, it requires that there is one and only one production that was applied first.

We now formalise these requirements. E1 expresses that $\alpha(\mathcal{E})$ partitions E_G. The MSO formula $\textsc{Part}_n(X,Y)$ defined in Equation 4.7 expresses that sets X and Y partition the domain. Here, we need to define a partition on a restricted domain so we define the formula $\textsc{ResPart}_n(Y,X_1,\ldots,X_n)$ to express that Y is partitioned by $X_1,\ldots,X_n$:

$$\textsc{ResPart}_n(Y,X_1,\ldots,X_n) : \forall y \in Y\Big(\big((y \in X_1) \vee \cdots \vee (y \in X_n)\big) \\ \wedge \big(\neg(y \in X_1 \wedge y \in X_2) \wedge \cdots \wedge \neg(y \in X_{n-1} \wedge y \in X_n)\big)\Big)$$

Recall that $\mathcal{W}$ is made up of edge variables $\mathcal{E}$ and anchor variables $\mathcal{C}$. We define E1 as:

$$\mathrm{E1}(\mathcal{W}) : \textsc{ResPart}_{|\mathcal{E}|}(E_G,\mathcal{E})$$

E2 expresses that for all $e \in \alpha(E_{i,p,j})$ e has label $\gamma_{p,j}$:

$$\mathrm{E2}(\mathcal{W}) : \bigwedge_{i,p,j} \forall e \in E_{i,p,j} \mathrm{lab}_{\gamma_{p,j}}(e).$$

Finally, E3 says there is a unique $p \in P_X$ such that $\alpha(E_{0,p,j})$ has exactly one element for each $j \in [|\mathrm{T}(p)|]$ and for every $p' \neq p$, $\alpha(E_{0,p',j})$ is empty for all j.

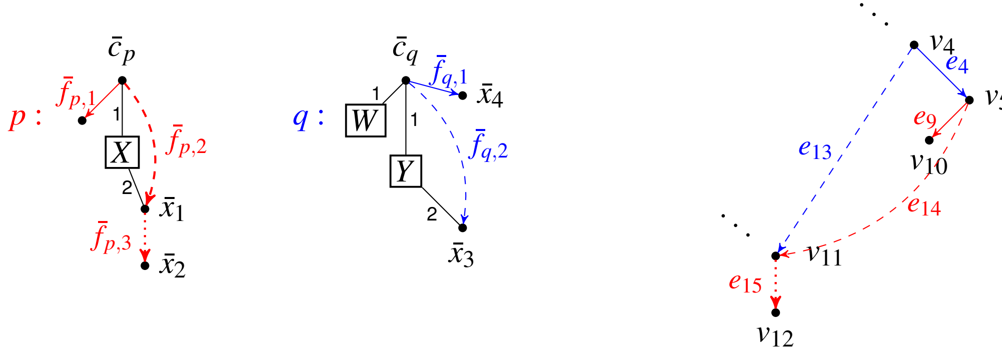
$$\mathrm{E3}_X(\mathcal{W}) : \bigvee_{p \in P_X} [\bigwedge_j \exists! e \in E_{0,p,j} \wedge \bigwedge_{p' \in P, p' \neq p} \bigwedge_j E_{0,p',j} = \emptyset]$$

where ! here as usual denotes uniqueness and the symbol $\wedge_{i,p,j}$ means we are quantifying over $i \in [0,|\mathrm{NT}(P)|]$, $p \in P$, and $j \in [|\mathrm{T}(p)|]$.

We then combine these formulas together to define $\textsc{EdgeSets}_X(\mathcal{W})$ as:

$$\textsc{EdgeSets}_X(\mathcal{W}) : \mathrm{E1}(\mathcal{W}) \wedge \mathrm{E2}(\mathcal{W}) \wedge \mathrm{E3}_X(\mathcal{W})$$

Let $\textsc{EdgeSets}(\mathcal{W}) = \textsc{EdgeSets}_S(\mathcal{W})$.



Example 32. Consider part of the grammar and graph from the running example shown above. The edges of the p production are shown in red, either as solid, dashed or dotted. The edges of the q production are shown in blue, as solid or dashed. The parameter $\mathcal{E}$ identifies the pre-image of each edge in this way.

For the whole graph shown in Figure 6.4, we obtain $\mathcal{E} = \{E_{0,p,1} = \{e_9\}, E_{0,p,2} = \{e_{14}\}, E_{0,p,3} = \{e_{15}\}, E_{1,q,1} = \{e_4, e_2\}, E_{1,q,2} = \{e_{13}, e_{11}\}, E_{2,r,1} = \{e_3\}, E_{2,r,2} = \{e_{12}\}, E_{2,s,1} = \{e_1\}, E_{2,s,2} = \{e_{10}\}, E_{1,t,1} = \{e_6, e_8\}, E_{2,u,1} = \{e_7\}, E_{1,u,1} = \{e_5\}\}$. This clearly forms a partition of the edges, the labels of the edges match those in the productions, $E_{0,p,i}$ has one element in it for $1 \leq i \leq 3$, and $E_{0,q,i}$ is empty for all $q \neq p$ so EDGESETS holds for this assignment.

Lemma 4. *Let $\mathcal{G}$ be an RGG and let $G \in \mathcal{L}(\mathcal{G})$ then for each derivation tree* $\mathbf{T}$ *of G,* $(G, \alpha_{\mathbf{T}}) \models$EDGESETS$(\mathcal{W})$.

Proof. We prove that for every $X \in N$ and every graph $G \in \mathcal{L}_X(\mathcal{G})$, if $\mathbf{T}$ is an X-derivation tree of G then $(G, \alpha_{\mathbf{T}}) \models$ EDGESETS$_X(\mathcal{W})$.

We prove by induction on the number of nodes in the derivation tree $\mathbf{T}$, denoted $|\mathbf{T}|$.

Base Case:

If $|\mathbf{T}| = 1$, then G is derived in one step, and $\mathbf{T}$ consists of a single node labelled q for some production $q : X \to G$ in P. We need to show that $(G, \alpha_{\mathbf{T}}) \models$ EDGESETS$_X(\mathcal{W})$ by showing that each of E1, E2 and E3 are satisfied. $G = \text{RHS}(q)$ so it is trivial to assign each edge in to a unique set $E_{0,q,j}$ for each $j \in [|\text{NT}(q)|]$. Therefore, $\mathcal{E}$ partitions E_G and so E1 is satisfied.

By the trivial assignment of each edge e to a set $E_{0,q,j}$ the label is maintained and $\text{lab}_{\gamma_{q,j}}(e)$ holds and E2 is satisfied.

Finally, E3 is satisfied since $q \in P_X$, each set $E_{0,q,j}$ for $j \in [|\text{T}(q)|]$ contains a single edge, and every other set $E_{i,p,j}$ is empty for $i \neq 0$ or $p \neq q$ so $E_{0,p,j} = \emptyset$ for each $p \neq q$

and j. Therefore, $(G, \alpha_{\mathbf{T}}) \models \text{EDGESETS}(\mathcal{W})$

Assumption:

Assume that if $|\mathbf{T}| < k$ and $G \in \mathcal{L}_X(\mathcal{G})$ then $(G, \alpha_{\mathbf{T}}) \models \text{EDGESETS}_X(\mathcal{W})$.

Inductive Step:

Let $|\mathbf{T}| = k$. G is in $\mathcal{L}_X(\mathcal{G})$ and so the root of $\mathbf{T}$ must have label q such that $q : X \to H$, H has n nonterminals $X_1, \ldots, X_n$ and $X_\eta \Rightarrow^* H_\eta$ for each $\eta \in [n]$ such that $H[X_1/H_1, \ldots, X_n/H_n] = G$. Then each H_η must have an X_η-derivation tree $\mathbf{T}_\eta$ such that $|\mathbf{T}_\eta| < k$ and so $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \text{EDGESETS}_{X_\eta}(\mathcal{W})$ for each $\eta \in [n]$. We need to prove that $(G, \alpha_{\mathbf{T}}) \models \text{EDGESETS}_X(\mathcal{W})$.

Recall the definition of $\alpha_{\mathbf{T}}$ with respect to $\alpha_{\mathbf{T}_\eta}$ in Equations 6.1 and 6.2. To show that $\alpha_{\mathbf{T}}(\mathcal{E})$ partitions E_G, we need to show that each e in E_G is assigned to a unique set $E_{i,p,j}$. If e is in E_{H_η} for some $\eta \in [n]$ then since $\alpha_{\mathbf{T}_\eta}(\mathcal{E})$ partitions the edges of E_{H_η}, e is still assigned to a unique $E_{i,p,j}$ (in some cases this will have changed from $E_{0,p,j}$ to $E_{\eta,p,j}$ but is still unique). If e is not in any E_{H_η} then it can be uniquely identified with an edge in E_H as in the base case. Therefore, E1 holds under $\alpha_{\mathbf{T}}$.

E2 holds for each edge in some E_{H_η} by the inductive hypothesis. For the edges in E_H, we uniquely identify each edge with an edge in the terminal subgraph of H and so the labels are preserved, $\text{lab}_{\gamma_{q,j}}(e)$ for each $e \in E_{0,q,j}$ for $j \in [|\text{T}(q)|]$. Therefore, E2 holds under $\alpha_{\mathbf{T}}$.

Finally, E3 says that there is a unique $p \in P_X$ such that $E_{0,p,j}$ has exactly one element for each j and for every $p' \neq p$ $E_{0,p',j}$ is empty for all j. Looking at the definition of $\alpha_{\mathbf{T}}$ in Equation 6.1, we can see that production q satisfies this requirement and $q \in P_X$ since $\text{LHS}(q) = X$. Every edge that was assigned to a set of the form $E_{0,p,j}$ by some $\alpha_{\mathbf{T}_\eta}$ has now been assigned to a set $E_{\eta,p,j}$ and so no edge is assigned to a triple that has $(0, p, j)$ for $p \neq q$ and so E3 is satisfied.

Therefore, $(G, \alpha_{\mathbf{T}}) \models \text{EDGESETS}_X(\mathcal{W})$ for $G \in \mathcal{L}_X(\mathcal{G})$ and for each $X \in N$. Therefore, $(G, \alpha_{\mathbf{T}}) \models \text{EDGESETS}(\mathcal{W})$ for $G \in \mathcal{L}(\mathcal{G})$.

□

6.4.4.2 Decomposition into subgraphs

This constraint partitions the graph into a set of connected subgraphs, each of which is isomorphic to the terminal subgraph of the right-hand side of some production. The MSO tests this isomorphism by looking at the paths from the anchor to each of the other nodes generated by the same production application. By (C3) of the definition

of RGG, there is a terminal and internal path from the anchor to every other node in a production. Using this fact and Lemma 3, we can uniquely identify the projections of these paths in a derived graph. We use a formula $\text{ANC}_{p,i,\bar{x}}(u,v,\mathcal{W})$ to express that there is some derivation tree node $\mathbf{v}$ such that $\mathbf{v}$ is an i-child, the label of $\mathbf{v}$ is p, $u = h(\bar{c}_p, \mathbf{v})$, and $v = h_v(\bar{x}, \mathbf{v})$. If this formula holds, we say that u **anchors** v. We first give a visual description of ANC and show how it is used in the precondition before formally defining its MSO statement.

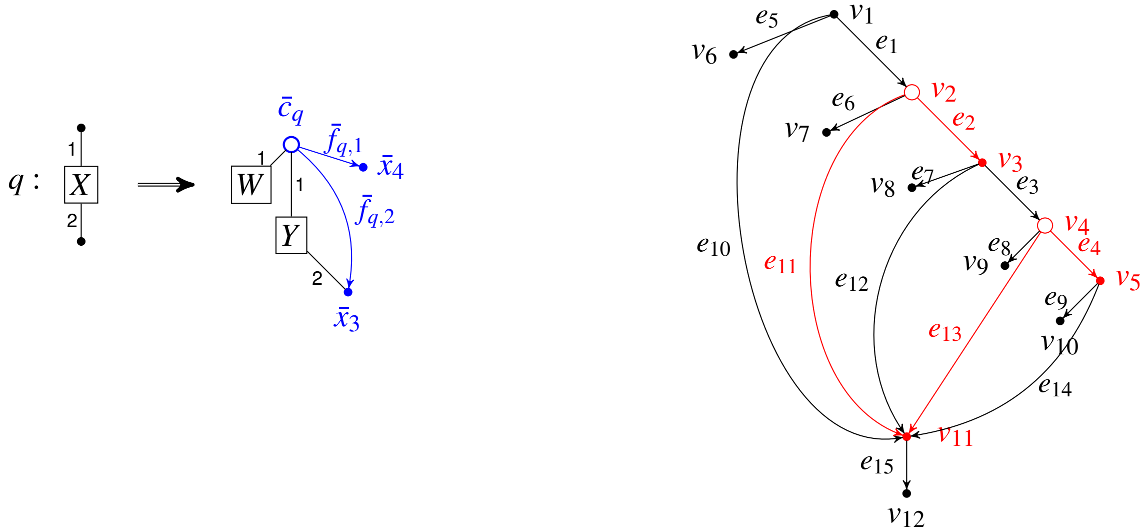


Figure 6.5: For each application of a production, we can relate the set of nodes generated by that production to the anchor.

Consider the anchor $\bar{c}_q$ shown as a hollow blue node on the left in Figure 6.5 above. This has two images, v_2 and v_4 in the graph, shown as hollow red nodes. The formula ANC will relate the anchor v_2 to the nodes generated by the same application of production q, and will do the same for v_4. This means that v_2 will be related to v_{11} and v_3; and v_4 will be related to v_{11} and v_5. Note that v_{11} is related to both anchors since it interacts with both productions.

Now we return to the precondition, which requires that:

(S1) Every node in G is attached to some edge,

(S2) for each anchor u we can identify a unique edge $e \in E_{i,p,j}$ for each $j \in |\text{T}(p)|$ such that u anchors all endpoints of e,

(S3) for each edge $e \in E_{i,p,j}$ we can identify a unique anchor u such that u anchors all endpoints of e.

We can write each of these requirements as MSO statements. We first define S1 as,

$\mathrm{S1} : \forall v \exists e \textsc{Inc}(e, v)$.

We define S2 and S3 in terms of a specific i, p, and j:

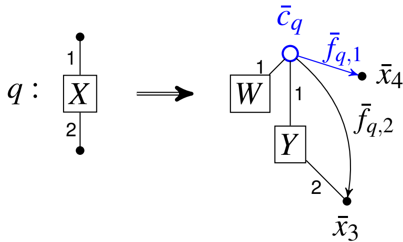
$$\mathrm{S2}_{i,p,j}(\mathcal{W}) : \forall c \in C_{i,p} \exists! e \in E_{i,p,j}$$
$$\exists v_1 \textsc{Anc}_{p,i,\bar{x}_1}(c, v_1, \mathcal{W}) \wedge \cdots \wedge \exists v_{j_k} \textsc{Anc}_{p,i,\bar{x}_{j_k}}(c, v_{j_k}, \mathcal{W}) \wedge \mathrm{edge}^2(e, v_1, \ldots, v_{j_k});$$

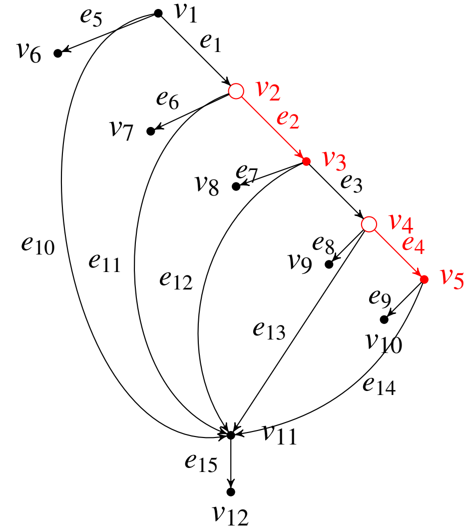
and

$$\mathrm{S3}_{i,p,j}(\mathcal{W}) : \forall e \in E_{i,p,j} \exists! c \in C_{i,p}$$
$$\exists v_1 \textsc{Anc}_{p,i,\bar{x}_1}(c, v_1, \mathcal{W}) \wedge \cdots \wedge \exists v_{j_k} \textsc{Anc}_{p,i,\bar{x}_{j_k}}(c, v_{j_k}, \mathcal{W}) \wedge \mathrm{edge}^2(e, v_1, \ldots, v_{j_k})$$

Then the formula $\textsc{Subgraph}$ is the conjunction of S1, S2, and S3 across all i, p, and j:

$$\textsc{Subgraph}(\mathcal{W}) : \mathrm{S1} \wedge \bigwedge_{i,p,j} \mathrm{S2}_{i,p,j} \wedge \mathrm{S3}_{i,p,j}$$





Example 33. In the figure above, we consider $\textsc{Subgraph}_{1,q,1}$. The anchor $\bar{c}_q$ and the edge $\bar{f}_{q,1}$ are highlighted in blue in the production on the left. The images of this anchor and edge are shown in red in the graph on the right.

We look at the anchors in $C_{1,q} = \{v_4, v_2\}$, and edges in $E_{1,q,1} = \{e_4, e_2\}$. The endpoints of e_4 are v_4 and v_5, and the endpoints of e_2 are v_2 and v_3. Looking at the graph, we can see that each of $\textsc{Anc}_{q,1,\bar{c}_q}(v_4, v_4, \mathcal{W})$, $\textsc{Anc}_{q,1,\bar{x}_4}(v_4, v_5, \mathcal{W})$, $\textsc{Anc}_{q,1,\bar{c}_q}(v_2, v_2, \mathcal{W})$,

and $\textsc{Anc}_{q,1,\bar{x}_4}(v_2, v_3, \mathcal{W})$ hold. Therefore, for $v_4 \in C_{1,q}$, the corresponding edge is $e_4 \in E_{1,q,1}$, and for $v_2 \in C_{1,q}$ it's $e_2 \in E_{1,q,1}$. Therefore, $\textsc{Subgraph}_{1,q,1}$ is satisfied.

We need to prove that each graph in $\mathcal{L}(\mathcal{G})$ satisfies the SUBGRAPH part of the precondition. Before we do so, we need to formally define the ANC formula.

Lemma 5 (Lemma 5.6 from Courcelle (1991))**.** *Let $\mathcal{G}$ be an RGG, G be a graph in $\mathcal{L}(\mathcal{G})$, and $\mathbf{T}$ be a derivation tree of G. For every $p \in P$, every $i \in [0, |NT(P)|]$, and every node $\bar{x} \in \textsc{rhs}(p)$, one can construct a formula $\textsc{Anc}_{p,i,\bar{x}}(u, w, \mathcal{W})$ such that, for every $u \in V_G \cup E_G$, $w \in V_G$:*

$$(G, u, w, \alpha_{\mathbf{T}}) \models \textsc{Anc}_{p,i,\bar{x}}(u, w, \mathcal{W})$$

iff $u = h(\bar{c}_p, \mathbf{v})$ and $w = h_v(\bar{x}, \mathbf{v})$ for some $\mathbf{v} \in V_{\mathbf{T}}$ which is an i-child and $p = lab_{\mathbf{T}}(\mathbf{v})$.

Proof. We need to deal with three cases: (1) where $\bar{c}_p$ is a node distinct from $\bar{x}$; (2) where $\bar{c}_p = \bar{x}$; and (3) where $\bar{c}_p$ is an edge (and therefore is distinct from $\bar{x}$).

Case 1: $\bar{c}_p$ is a node distinct from $\bar{x}$. By C3 of RGG, there is a terminal and internal path π from $\bar{c}_p$ to $\bar{x}$. We fix this path to be

$$(\bar{c}_p, \bar{f}_{p,m_1}, i_1, j_1, \bar{v}_1) \ldots (\bar{v}_{k-1}, \bar{f}_{p,m_k}, i_k, j_k, \bar{x}).$$

Let $\mathrm{tr}(\pi)$ be the trace of this path:

$$((i, p, m_1), i_1, j_1) \ldots ((i, p, m_k), i_k, j_k).$$

Let $\bar{f}_{p,m}$ be an edge of $\textsc{rhs}(p)$ such that $\textsc{Inc}_\eta(\bar{f}_{p,m}, \bar{c}_p)$. In the formula, we will require that u is an anchor. By specifying that it must be connected to a specific tentacle of an edge in $E_{i,p,m}$, we establish that it is an anchor for an i-child labelled by p. Let $\textsc{Anc}^1_{p,i,\bar{x}}(u, w, \mathcal{W})$ be a formula expressing the following facts:

F1 $u \in \mathcal{C}$ and u is the η-th node of an edge in $E_{i,p,m}$

F2 there is a path with trace $\mathrm{tr}(\pi)$ from u to w

To express that, for some edge e in a path, one has $\mathrm{o}(e) = (i, p, j)$, it suffices to write $e \in E_{i,p,j}$. The MSO formula is based on $\mathrm{tr}(\pi)$, the trace of the path from $\bar{c}_p$ to $\bar{x}$, and the fact that $\bar{c}_p$ is the η-th node of $\bar{f}_{p,m}$.

$$\begin{aligned}
&\textsc{Anc}^1_{p,i,\bar{x}}(u, w, \mathcal{W}) : \\
&\exists e \in E_{i,p,m} \textsc{Inc}_\eta(e, u) \land u \in \mathcal{C} \\
&\land \exists e_1 \ldots e_k v_1 \ldots v_{k-1} \Big((e_1 \in E_{i,p,m_1} \land \textsc{Inc}_{i_1}(e_1, u) \land \textsc{Inc}_{j_1}(e_1, v_1)) \\
&\qquad \land \cdots \land (e_k \in E_{i,p,m_k} \land \textsc{Inc}_{i_k}(e_k, v_{k-1}) \land \textsc{Inc}_{j_k}(e_k, w)) \Big)
\end{aligned}$$

If: If $u = h_v(\bar{c}_p, \mathbf{v})$ for $\mathbf{v}$ an i-child with $p = lab_{\mathbf{T}}(\mathbf{v})$ then $u \in \mathcal{C}$ by the definition of $\mathcal{C}$ and if $\bar{c}_p$ is the η-th node of $\bar{f}_{p,m}$ then $\textsc{Inc}_\eta(e,u)$ for some $e \in E_{i,p,m}$ by the definition of $\mathcal{E}$. If $w = h_v(\bar{x}, \mathbf{v})$ then F2 holds for path $h(\pi, \mathbf{v})$ by Lemma 3.

Only if: Let conversely u, w satisfy $\textsc{Anc}_{p,i,\bar{x}}(u, w, \mathcal{W})$. Then we need to show that F1 and F2 being satisfied imply that $u = h(\bar{c}_p, \mathbf{v})$ and $w = h_v(\bar{x}, \mathbf{v})$ for some $\mathbf{v}$ which is an i-child. Let λ be a path satisfying F2. By F1, $u \in \mathcal{C}$ and u is the η-th node of an edge $e \in E_{i,p,m}$, $\bar{c}_p$ is the η-th node of $\bar{f}_{p,m}$ in $\textsc{rhs}(p)$ and so $u = h(\bar{c}_p, \mathbf{v})$ for some $\mathbf{v}$ such that $\text{lab}_{\mathbf{T}}(\mathbf{v}) = p$ and $\mathbf{v}$ is an i-child. The path π in $\textsc{rhs}(p)$ links $\bar{c}_p$ to $\bar{x}$ and so by Lemma 3, $h(\pi, \mathbf{v})$ links $h(\bar{c}_p, \mathbf{v})$ to $h(\bar{x}, \mathbf{v})$ with trace $\text{tr}(\pi)$. By F2, there is a also a path with trace $\text{tr}(\pi)$ from u to w. Since $u = h(\bar{c}_p, \mathbf{v})$ and every node on the path (except possibly the last node) are internal, Lemma 3 says that the path from u to w is identical to the path from $h(\bar{c}_p, \mathbf{v})$ to $h(\bar{x}, \mathbf{v})$. Therefore $w = h(\bar{x}, \mathbf{v})$.

Case 2: $\bar{x} = \bar{c}_p$. We define $\textsc{Anc}^2_{p,i,\bar{x}}$ in this case expressing that $u \in \mathcal{C}$, $u = w$, and u is the η-th node of some edge in $E_{i,p,m}$, as in F1. This is written as:

$$\textsc{Anc}^2_{p,i,\bar{c}_p}(u, w, \mathcal{W}) : u \in \mathcal{C} \wedge u = w \wedge \exists e \in E_{i,p,m} \textsc{Inc}_\eta(e, u).$$

It is clear that if $\bar{x} = \bar{c}_p$ then this holds. In the opposite direction, $u \in \mathcal{C}$ means that by Lemma 2 if $u = w$ and u and w are not images of the same node, then w must be the image of an external node. However, $w \in \mathcal{C}$ means that this is impossible and so w and u must be the image of the same node, and so $\bar{c}_p = \bar{x}$.

Case 3: $\bar{c}_p$ is an edge. The only case in which this happens is if p consists of a single terminal edge. Therefore, w here will be a node attached to the anchor edge u. If $\textsc{Inc}_j(\bar{c}_p, \bar{x})$ then

$$\textsc{Anc}^3_{p,i,\bar{x}}(u, w, \mathcal{W}) : u \in E_{i,p,1} \wedge \textsc{Inc}_j(u, w).$$

This is clearly true if and only if $\bar{x}$ is the j-th node of $\bar{c}_p$ and $u = h(\bar{c}_p, \mathbf{v})$ where $p = \text{lab}(\mathbf{v}) \in \text{ext-}P$ and $\mathbf{v}$ is an i-child.

Finally, we define:

$$\textsc{Anc}_{p,i,\bar{x}}(u, w, \mathcal{W}) : \quad \textsc{Anc}^1_{p,i,\bar{x}}(u, w, \mathcal{W}) \vee \textsc{Anc}^2_{p,i,\bar{x}}(u, w, \mathcal{W}) \vee \textsc{Anc}^3_{p,i,\bar{x}}(u, w, \mathcal{W}).$$

□

We use the fact that a node or edge anchors itself to establish its corresponding production. We write this as $u \in C_{i,p}$ indicating that u is an anchor for an i-child with label p. Formally for each $p \in P$, and each $i \in [0, |\text{NT}(P)|]$,

$u \in C_{i,p} \Leftrightarrow \textsc{Anc}_{p,i,\bar{c}_p}(u,u,\mathcal{W})$.

We can consider the set of $C_{i,p}$ sets to be a more fine-grained version $\mathcal{C}$ defined above, and that $\mathcal{C} = \bigcup_{i,p} C_{i,p}$. Recall Equations 6.1 and 6.2 which defines $\alpha_{\mathbf{T}}$ in terms of its subtrees $\alpha_{\mathbf{T}_\eta}$. As a consequence of these equations, we get that if some anchor u lies in a set $C_{0,p}$ under $\alpha_{\mathbf{T}_\eta}$, then under $\alpha_{\mathbf{T}}$ it lies in $C_{\eta,p}$.

Example 34. Looking again at Figure 6.5, we can see that there are two images of production q in the graph. The nodes v_2, v_3 and v_{11} are anchored by v_2; and v_5, v_4 and v_{11} are anchored by v_4. Formally for v_2: $\textsc{Anc}_{q,1,\bar{c}_q}(v_2,v_3)$, $\textsc{Anc}_{q,1,\bar{c}_q}(v_2,v_{11})$, and $\textsc{Anc}_{q,1,\bar{c}_q}(v_2,v_2)$ hold, implying that $v_2 \in C_{1,q}$.

Now that we have defined the MSO formula for $\textsc{Anc}$, we can use it to show that any graph in the language satisfies the $\textsc{Subgraph}$ part of the precondition. We leave the proof of the following lemma to the appendix as it has a similar structure to the proof of Lemma 4.

Lemma 6. *Let $\mathcal{G}$ be an RGG and let $G \in \mathcal{L}(\mathcal{G})$ then for each derivation tree $\mathbf{T}$ of G, $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$.*

We prove by induction over the size of the derivation tree. As in Lemma 4, we prove that for every $G \in \mathcal{L}_x(\mathcal{G})$, $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$. Again, we use the construction of $\alpha_{\mathbf{T}}$ as defined in Equations 6.1 and 6.2.

6.4.5 RGLs satisfy the precondition

The precondition of the transducer is the conjunction of each of these formulas,

$\rho_X(\mathcal{W}) : \textsc{EdgeSets}_X(\mathcal{W}) \wedge \textsc{Subgraph}(\mathcal{W})$

Define $\rho(\mathcal{W}) = \rho_S(\mathcal{W})$.

Proposition 3. *Let $\mathcal{G}$ be an RGG and let $G \in \mathcal{L}(\mathcal{G})$, then for each derivation tree $\mathbf{T}$ of G, there exists a parameter assignment $\alpha_{\mathbf{T}}$ such that $(G, \alpha_{\mathbf{T}}) \models \rho(\mathcal{W})$.*

Proof. We use the parameter assignment $\alpha_{\mathbf{T}}$ which is defined from $\mathbf{T}$ in §6.4.1. Lemma 4 proves that $(G, \alpha_{\mathbf{T}}) \models \textsc{EdgeSets}(\mathcal{W})$. Lemma 6 proves that $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$. Therefore, $(G, \alpha_{\mathbf{T}}) \models \rho(\mathcal{W})$. □

6.5 Parsing as transduction

The transducer is made up of three types of formulas: the precondition, the domain formulas, and the relation formulas. We have established the precondition $\rho(\mathcal{W})$ and next we define the domain and relation formulas.

6.5.1 The formulas of the transducer

The domain formulas define the nodes of the derivation tree and so we write $\mathrm{node}(x, \mathcal{W})$. The relation formulas define which output node is the i-th child of another output node, written $\mathrm{child}_i(x, y, \mathcal{W})$, and the labels of the output nodes, written $\mathrm{lab}_p(x, \mathcal{W})$.

The domain of the output for a parameter assignment α is $D_{\mathbf{T}}$ where:

$$D_{\mathbf{T}}(\alpha) : \{x \mid (G, x, \alpha) \models \mathrm{node}(x, \mathcal{W})\}$$

and $\mathrm{node}(x, \mathcal{W}) : x \in \mathcal{C}$. This means that the nodes of the derivation tree are simply generated by the anchors in the graph. This relies on the requirement of RGGs that there is always an internal node generated by each production (besides those with a single terminal edge and all external nodes).

The relation formula $\mathrm{child}_i(x, y, \mathcal{W})$ defines the edges of the output of the transducer. We use the formula $\textsc{Par}_{p,i,p',i'}(u, u', \mathcal{W})$, which we will define below, that encodes that the derivation tree node corresponding to u' is the i'-th child of the node corresponding to u (which itself is the i-th child of some other node).

$$\mathrm{child}_{i'}(x, y, \mathcal{W}) : \bigvee_{i,p,p'} (\textsc{Par}_{p,i,p',i'}(x, y, \mathcal{W})$$

We also need to assign labels to the tree nodes. The labels of the derivation tree correspond to the labels of the production, this can be done via the unary relation:

$$\mathrm{lab}_p(x, \mathcal{W}) : \bigvee_i x \in C_{i,p}.$$

We now formally define the formula $\textsc{Par}$ encoding the parent-child relationships in the output derivation tree.

Specifically, $\textsc{Par}$ will relate one anchor u to another u' if u and u' correspond to derivation tree nodes $\mathbf{v}$ and $\mathbf{v}'$ and $\mathbf{v}$ is the parent of $\mathbf{v}'$. Consider Figure 6.6, showing part of Figure 6.4. The subgraph in red shows the image of production p, and the blue shows the image of the first application of production q (at derivation tree node $\mathbf{v}_2$). The anchors of each production are shown as hollow nodes in the graph. The formula $\textsc{Par}$ will establish a relationship between v_5—the anchor of the p subgraph, and v_4—

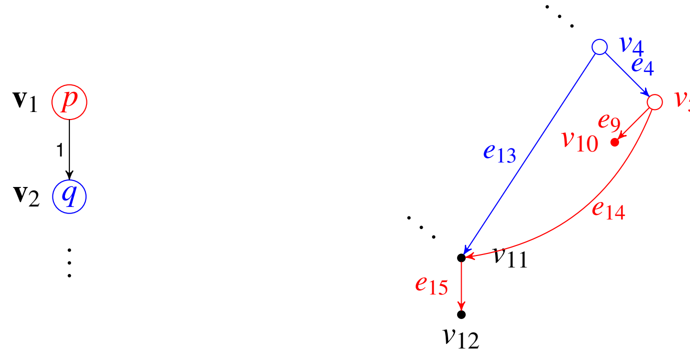


Figure 6.6: Part of Figure 6.4 showing the images of productions p and q in the graph and how they interact. PAR will establish the the anchor v_4 should be a child of the anchor v_5.

the anchor of the q subgraph. The formula does so by checking that the subgraphs are connected in a way that is allowed by the grammar.

We leave the proof of the following lemma to the appendix as the mechanics are similar to that of the proof of Lemma 5.

Lemma 7 (Lemma 5.7 of Courcelle (1991))**.** *Let $\mathcal{G}$ be an RGG, G be in $\mathcal{L}(\mathcal{G})$, $\mathbf{T}$ be a derivation tree of G, and α be the parameter assignment defined with respect to $\mathbf{T}$. One can construct a formula $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$ such that, for $u,w \in V_G \cup E_G$:*

$$(G,u,w,\alpha) \models \text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$$

iff $u = h(\bar{c}_p, \mathbf{v}), w = h(\bar{c}_{p'}, \mathbf{v}')$ for some $\mathbf{v}, \mathbf{v}'$ in $V_\mathbf{T}$ where $p = lab_\mathbf{T}(\mathbf{v})$, $p' = lab_\mathbf{T}(\mathbf{v}')$, $\mathbf{v}$ is an i-child, and $\mathbf{v}'$ is the i'-th child of $\mathbf{v}$ in $\mathbf{T}$.

While we do not provide the full proof in the text here, we give the formula constructed for PAR and the intuitions behind it. To establish that $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$ holds, we need to identify the set of nodes shared by the two productions—the nodes attached to the i'-th nonterminal of p and the external nodes of p'.

Let $\bar{x}_1,\ldots,\bar{x}_k$ be the sequence of nodes of the i'-th nonterminal edge of $\text{RHS}(p)$ and $\bar{y}_1,\ldots,\bar{y}_k$ be the sequence of external nodes of $\text{RHS}(p')$. Then in the formula we establish the nodes $v_1,\ldots,v_k$ in the graph such that each v_j is the image of both $\bar{x}_j$ and $\bar{y}_j$. We do this using the formula ANC from Lemma 5.

$$\begin{aligned}&\textsc{Par}_{p,i,p',i'}(u,w,\mathcal{W}):\\&\exists v_1,\ldots,v_k\Big(\textsc{Anc}_{p,i,\bar{x}_1}(u,v_1,\mathcal{W})\wedge\textsc{Anc}_{p',i',\bar{y}_1}(w,v_1,\mathcal{W})\wedge\cdots\wedge\\&\qquad\textsc{Anc}_{p,i,\bar{x}_k}(u,v_k,\mathcal{W})\wedge\textsc{Anc}_{p',i',\bar{y}_k}(w,v_k,\mathcal{W})\Big).\end{aligned}$$

If $\textsc{Par}_{p,i,p',i'}(u,u',\mathcal{W})$ holds, then u will become the parent of u' in the output tree. The proof of this lemma relies on C2 of RGG. Without C2, we could not say for sure that one node should be a parent or a child of another in the output tree.

Example 35. From Figure 6.6, the anchor of the p production is v_5 and the anchor of the q production is v_4, therefore $\textsc{Par}_{p,0,q,1}(v_5,v_4,\mathcal{W})$ holds.

We now have each part of the transducer and can consider what the output might look like for a given graph.

Example 36. Figure 6.7 shows the output of the transducer when it takes Figure 6.4 as input with α defined as in the previous examples. The domain formulas specify the existence of the 9 nodes and the relation formulas specify the edges between the nodes, labelled by $\textsc{Par}$ formulas, and the labels of the nodes, according to the $C_{i,p}$ sets. We have the formulas written on the tree to show where the labels and edges come from.

Let $\mathcal{G}$ be an RGG, and let $X \in N$. Then the corresponding transducer τ_X is
$\langle \rho_X(\mathcal{W}), \text{node}(x,\mathcal{W}), (\text{lab}_p(x,\mathcal{W}))_{p\in P}, (\text{child}_i(x,y,\mathcal{W}))_{i\in[|\text{NT}(P)|]}\rangle$.

For start symbol S of $\mathcal{G}$, let $\tau = \tau_S$. Let G be a graph in $\mathcal{L}(\mathcal{G})$, and let α be a parameter assignment such that $(G,\alpha) \models \rho(\mathcal{W})$. Then the output of the transducer with respect to α is

$$\tau(G,\alpha) = (V_H, \text{lab}_H, (\text{child}^i_H)_{i\in[0,|\text{NT}(P)|]}),$$

where $V_H = D_{\mathbf{T}}(\alpha) = \{x \mid (G,x,\alpha) \models \text{node}(x,\mathcal{W})\}$, $\text{lab}_H : V_H \to P$ such that $\text{lab}_H(x) = p$ if $x \in \alpha(C_{i,p})$ for some i, and $\text{child}^i_H : V_H \to V_H$ such that $\text{child}^H_i(x,y)$ if $(G,x,y,\alpha) \models \textsc{Par}_{p,i,p',r}(x,y,\mathcal{W})$.

6.5.2 Transducer output and derivation trees

This section sets us up to use the backwards translation theorem to show that $\mathcal{L}(\mathcal{G})$ is MSO-definable. We first sketch what the proof will look like and then fill in the parts necessary to prove it. Recall that the backwards translation theorem states that if L is MSO-definable, and τ is an MSO transducer then $\tau^{-1}(L)$ is MSO-definable.

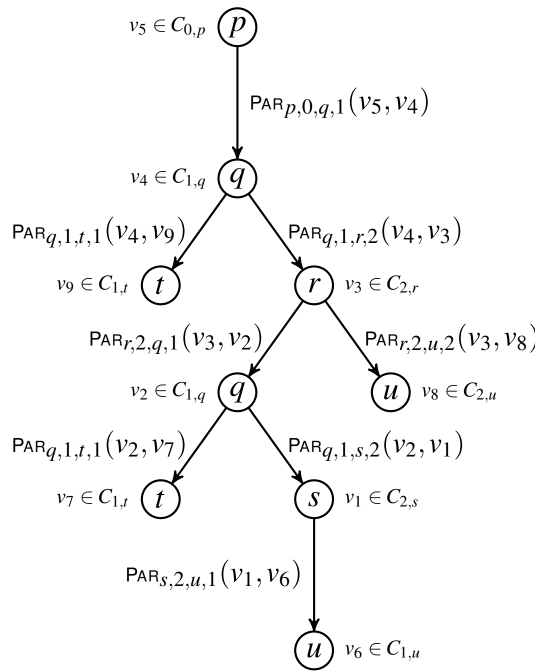


Figure 6.7: The output of the transducer, variable names are based on those of Figure 6.4. The PAR formulas are there to explain why the edge exists and the $v \in C_{i,p}$ formulas are there to show where the node labels come from.

As we have mentioned many times in this chapter, we will use the fact that the derivation trees of any HRG (and therefore RGG) are MSO-definable. For an RGG $\mathcal{G}$, let $\mathcal{T}_\mathcal{G}$ be its set of derivation trees. We can use the backwards translation theorem to show that $\tau^{-1}(\mathcal{T}_\mathcal{G})$ is MSO-definable. It remains to show that $\tau^{-1}(\mathcal{T}_\mathcal{G}) = \mathcal{L}(\mathcal{G})$.

Formally,

$$\tau^{-1}(\mathcal{T}_\mathcal{G}) = \{G \mid \tau(G) \cap \mathcal{T}_\mathcal{G} \neq \emptyset\}. \tag{6.3}$$

So we aim to show that

$$\mathcal{L}(\mathcal{G}) = \{G \mid \tau(G) \cap \mathcal{T}_\mathcal{G} \neq \emptyset\}.$$

We do so in two steps, Proposition 4 shows that for each $G \in \mathcal{L}(\mathcal{G})$, if $\mathbf{T}$ is a derivation tree of G then $\mathbf{T} \in \tau(G)$. This will prove that

$$\mathcal{L}(\mathcal{G}) \subseteq \{G \mid \tau(G) \cap \mathcal{T}_\mathcal{G} \neq \emptyset\}.$$

Proposition 5 shows the opposite inclusion. It proves that for arbitrary G, if $\mathbf{T} \in \tau(G)$ is in $\mathcal{T}_\mathcal{G}$ then $G \in \mathcal{L}(\mathcal{G})$, and therefore:

$$\{G \mid \tau(G) \cap \mathcal{T}_\mathcal{G} \neq \emptyset\} \subseteq \mathcal{L}(\mathcal{G}).$$

Proposition 4. *Let $\mathcal{G}$ be an RGG and τ be the corresponding transducer. Let $G \in \mathcal{L}(\mathcal{G})$ and $\mathbf{T}$ be a derivation tree of G. Then $\mathbf{T} \in \tau(G)$.*

Proof. We show that for each $X \in N$, if $G \in \mathcal{L}_X(\mathcal{G})$ with X-derivation tree $\mathbf{T}$, then $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$.

By Proposition 3, if $G \in \mathcal{L}_X(\mathcal{G})$ with X-derivation tree $\mathbf{T}$ then $(G, \alpha_{\mathbf{T}}) \models \rho_X$. So now we must show that the output of the transducer is $\mathbf{T}$. We prove by induction on $|\mathbf{T}|$.

Base Case:

$G \in \mathcal{L}_X(\mathcal{G})$, $(G, \alpha_{\mathbf{T}}) \models \rho_X(\mathcal{W})$ and $|\mathbf{T}| = 1$. By $\rho_X(\mathcal{W})$, every edge in G is assigned to a set of the form $E_{i,p,j}$ for $i \in [0, |\mathrm{NT}(P)|]$, $p \in P$, and $j \in [|\mathrm{T}(p)|]$. Let u be the single anchor and assume that $u \in C_{i,p}$ for some i and p, then since there is only one anchor each edge must be in a set of the form $E_{i,p,j}$ for fixed i and p and $j \in [|\mathrm{T}(p)|]$. By EDGESETS, there must be some set of edges in sets of the form $E_{0,q,j}$ for $q \in P_X$ (such that q is the label of the root of $\mathbf{T}$) and $j \in [|\mathrm{T}(q)|]$. Therefore, for each edge in G, there is a unique $j \in [|\mathrm{T}(q)|]$ such that $e \in E_{0,q,j}$. By SUBGRAPH, u is in $C_{0,q}$. In terms of the output formulas of τ_X, $\mathrm{node}(u, \mathcal{W})$ holds and $\mathrm{lab}_q(u)$ holds. Therefore, the output of $\tau_X(G, \alpha_{\mathbf{T}})$ is a single node labelled q. The root of $\mathbf{T}$ is q and $\mathbf{T}$ has no other nodes so $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$. This implies that for each X-derivation tree $\mathbf{T}$ of G of size 1, $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$.

Assumption:

Assume that if $G \in \mathcal{L}_X(\mathcal{G})$ and $|\mathbf{T}| < k$ then $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$.

Inductive Case:

Let $G \in \mathcal{L}_X(\mathcal{G})$ and let $\mathbf{T}$ be a derivation tree of G such that $|\mathbf{T}| = k$, we need to show that $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$. Let $q \in P_X$ be the label of the root of $\mathbf{T}$ and let $q : X \to H$ be such that H has nonterminals $Y_1, \ldots, Y_n$. Then $G = H[Y_1/H_1] \ldots [Y_n/H_n]$ for $H_\eta \in \mathcal{L}_{Y_\eta}(\mathcal{G})$ for each $\eta \in [n]$. Let $\mathbf{T}_1, \ldots, \mathbf{T}_n$ be the subtrees of $\mathbf{T}$ such that for each $\eta \in [n]$, the root of $\mathbf{T}_\eta$ is a child of the root of $\mathbf{T}$. For each $\eta \in [n]$, $\mathbf{T}_\eta$ is a Y_η-derivation tree of H_η and $|\mathbf{T}_\eta| < k$. Therefore, for each $\eta \in [n]$, $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \rho_{Y_\eta}(\mathcal{W})$ and $\tau_{Y_\eta}(H_\eta, \alpha_{\mathbf{T}_\eta}) = \mathbf{T}_\eta$. Using the relationship between $\alpha_{\mathbf{T}}$ and each $\alpha_{\mathbf{T}_\eta}$ as defined in Equations 6.1 and 6.2, the output of $\tau_X(G, \alpha_{\mathbf{T}})$ is going to be the set of subtrees $\mathbf{T}_1, \ldots, \mathbf{T}_n$ along with a node labelled q. Using the definition of PAR, $\mathrm{child}_\eta(u, u_\eta, \mathbf{T})$ for u the single node with label q and u_η the root of $\mathbf{T}_\eta$ for each $\eta \in [n]$. Therefore, $\tau_X(G, \alpha_{\mathbf{T}}) = \mathbf{T}$.

This holds for each $X \in N$, so by letting $X = S$ we get that for each $G \in \mathcal{L}(\mathcal{G})$ with derivation tree $\mathbf{T}$, $\tau(G, \alpha_{\mathbf{T}}) = \mathbf{T}$. Therefore, $\mathbf{T} \in \tau(G)$. □

By Proposition 4, we know that for each G, $\{\mathbf{T} \mid \mathrm{VAL}(\mathbf{T}) = G\} \subseteq \tau(G)$. The next

proposition will establish the inclusion in the opposite direction, that $\tau(G) \subseteq \{\mathbf{T} \mid \textsc{val}(\mathbf{T}) = G\}$. As we did with earlier lemmas, we leave the proof of the following proposition to the appendix.

Proposition 5. *Let $\mathcal{G}$ be an RGG and G be a graph not necessarily in $\mathcal{L}(\mathcal{G})$. Let α be a parameter assignment such that $(G,\alpha) \models \rho(\mathcal{W})$. Then if $\mathbf{T} = \tau(G,\alpha)$ is in $\mathcal{T}_{\mathcal{G}}$ then $\textsc{val}(\mathbf{T}) = G$ and so $G \in \mathcal{L}(\mathcal{G})$.*

We prove that for each $X \in N$, if $(G,\alpha) \models \rho(\mathcal{W})$ and $\mathbf{T} \in \tau_X(G,\alpha)$ is an X-derivation tree, it is an X-derivation tree of G, implying that $G \in \mathcal{L}_X(\mathcal{G})$. We prove this by induction on the size of the anchor set, $\mathcal{C}$. In the inductive step, we show that the graph G can be decomposed into a set of connected subgraphs $G', H_1, \ldots, H_n$. We then show that the subgraph G' corresponds to the terminal subgraph of $\textsc{rhs}(q)$ in some production $q \in P_X$; and for each $i \in [n]$, $Y_i \Rightarrow^* H_i$ where X_i lies in $\textsc{rhs}(q)$.

Now that we have established Propositions 4 and 5, we can complete the proof that RGL are MSO-definable.

Theorem 5. *RGL $\subseteq$ MSO graph languages.*

Proof. Let $\mathcal{G}$ be an RGG and $\mathcal{T}_{\mathcal{G}}$ be its set of derivation trees. Let τ be the MSO transducer as described with respect to $\mathcal{G}$.

Recall that

$$\tau^{-1}(\mathcal{T}_{\mathcal{G}}) = \{G \mid \tau(G) \cap \mathcal{T}_{\mathcal{G}} \neq \emptyset\}.$$

By Proposition 4,

$$\mathcal{L}(\mathcal{G}) \subseteq \{G \mid \tau(G) \cap \mathcal{T}_{\mathcal{G}} \neq \emptyset\}.$$

By Proposition 5,

$$\{G \mid \tau(G) \cap \mathcal{T}_{\mathcal{G}} \neq \emptyset\} \subseteq \mathcal{L}(\mathcal{G}).$$

Therefore,

$$\tau^{-1}(\mathcal{T}_{\mathcal{G}}) = \mathcal{L}(\mathcal{G})$$

and so by the backwards translation theorem, $\mathcal{L}(\mathcal{G})$ is MSO-definable. □

6.6 RGLs are closed under intersection

MSO graph languages are trivially closed under intersection, since the conjunction of two MSO statements is also an MSO statement: if φ_1 and φ_2 are both MSO formulae,

then $\mathcal{L}(\varphi_1) \cap \mathcal{L}(\varphi_2) = \mathcal{L}(\varphi_1 \wedge \varphi_2)$. However, this does not guarantee that an arbitrary subfamily of MSO graph languages is closed under intersection—it only guarantees that the intersection of two languages in a subfamily is an MSO graph language, but is not necessarily in the subfamily itself. Here, we give a sufficient condition for a subfamily $\mathbb{F}$ to be closed under intersection.

Theorem 6. *Let $\mathbb{G}$ be (1) a subfamily of HRL, defined as a restriction on the right-hand sides of its productions that does not depend on nonterminal labels; and (2) let the family of languages $\mathbb{F}$ generated by $\mathbb{G}$ be MSO-definable. Then for any pair of languages $L_1, L_2 \in \mathbb{F}$, the language $L_1 \cap L_2$ is also in $\mathbb{F}$.*

Proof. Since both L_1 and L_2 are both in HRL and the MSO graph languages, we can look at them from both perspectives. Let $\mathcal{G}_1$ be a HRG deriving L_1 and let ϕ_2 be an MSO statement defining L_2. Propositions 1.10 and 4.8 in Courcelle (1990) prove that the intersection of a HR language and an MSO language is in HRL, by constructing a HRG which derives all and only those graphs in the intersection of the two languages.[4] This HRG has finitely many nonterminals defined by the cross product of the nonterminals of the original HRG and a finite set of 'states' of the MSO.[5] The productions of the intersection grammar are copies of the original HRG, with different nonterminal labels. Hence we can construct HRG $\mathcal{G}_\cap$ such that $\mathcal{L}(\mathcal{G}_\cap) = L_1 \cap L_2$ and the productions in $\mathcal{G}_\cap$ satisfy any restriction that $\mathcal{G}_1$ satisfied since the restriction is in terms of the right-hand sides of the productions but not the nonterminal labels. Therefore, $\mathcal{G}_\cap$ is in $\mathbb{G}$. □

RGG satisfies the conditions of Proposition 6, so RGLs are closed under intersection. Importantly, both proofs—that RGLs are MSO-definable and closed under intersection—are constructive, implying that it is possible to construct the intersection grammar.

[4]This is a generalisation to graphs of the proof that the intersection of a context-free and regular string language is a context-free string language Bar-Hillel et al. (1961).

[5]For each MSO-definable language L in some set of all possible graphs L', there exists a set A, a homomorphism $h : L' \to A$, and a finite subset C of A such that $L = h^{-1}(C)$. The finite set of 'states' here is the set C.

6.7 Conclusions

The grammar used as the running example, introduced in Table 6.1, is rather unnatural in how it models control—it generates the structure bottom-up. It would be more natural to derive the graph in Figure 6.3 from outermost to innermost predicate; but constraint C3 makes it difficult to express this, and the grammar in Table 6.1 does not. Remark 2 also discusses how we can no longer use unary edges to represent node labels as we did in Chapter 5 for HRG. These difficulties arise from the restrictive definition of RGG and lead to a desire for a more flexible formalism that is still MSO-definable.

As we mentioned, an example of a language that is in SCFL but cannot be generated by RGG is the BEACHBALL language. If we were to allow this language into RGL, then in the context of the proof, PAR would no longer hold as there is no way of identifying which order the edges are generated in. Courcelle (1991) discusses this problem and introduces an alternative representation of derivation trees called **reduced trees** which enable some cases of this type to be defined in MSO logic. This point requires further investigation.

Another possible extension of RGG would be to consider alternative forms of Lemma 3. Every MSO formula in the transducer depends on this lemma. RGG could potentially be extended if other cases in which a path could be defined in terms of its trace and initial node. In the next chapter, we will examine two other formalisms which appear to be related to RGG: the restricted DAG grammars (Björklund et al., 2016), and the tree-like grammars (Matheja et al., 2015). We will compare these formalisms to RGG in terms of their expressivity.

Chapter 7

Comparing RGG to other formalisms

Recall from Chapter 6 that Courcelle (1991) defined the strongly context-free languages (SCFL) as the family of languages that are contained both within the MSO graph languages and the HR languages. We know that RGL are contained within SCFL. However, there are other families of graph languages in this space which we want to compare to RGL. The first is the tree-like languages (TLL; Matheja et al. 2015), defined using a restricted form of HRG, which are MSO-definable. This implies that they are also a subfamily of SCFL. The second family is the restricted DAG languages (RDL; Björklund et al. 2016) which are also a restricted form of HRG. However, they have not been shown to be MSO-definable but we believe it is likely that they are, due to the fact it is possible to deterministically extract the derivation structure from an RDL graph.

7.1 Restricted DAG grammars

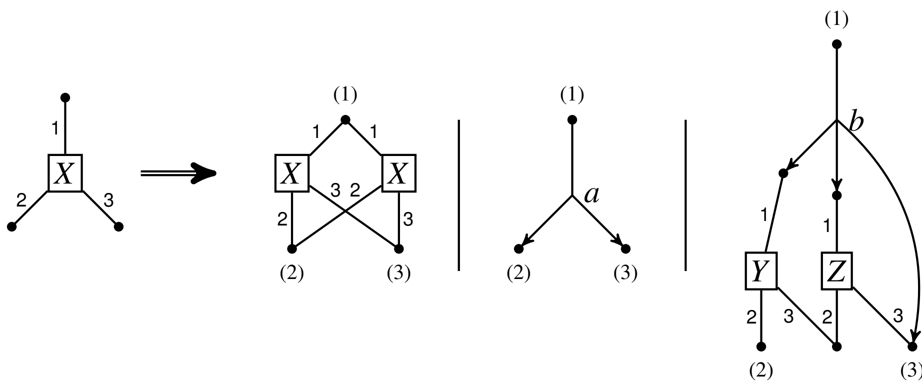
Restricted DAG Grammars (RDGs) operate over single-rooted ordered DAGs where each edge has a single source node and any number of target nodes. By ordered, we mean that the target nodes of each edge is an ordered sequence, this is different to ordered DAG automata which generate only planar graphs. Formally, edges are attached to nodes using the functions src: $E \to V$, and tar: $E \to V^*$.

For simplicity, we define RDGs by looking at their normal form productions. Björklund et al. (2016) show that every RDG has such a normal form. Each production $X \to G$ in the normal form is in one of three possible forms:

- $X \to G$ is a clone: G consists of two edges, e_1 and e_2, such that $\mathrm{src}(e_1) = \mathrm{src}(e_2)$, $\mathrm{tar}(e_1) = \mathrm{tar}(e_2)$, $\mathrm{lab}(e_1) = \mathrm{lab}(e_2) = X$.

- G is a single terminal edge.
- Or:
  - the longest directed path in G is of length 2;
  - there is exactly one edge e whose source is the root and e is terminal;
  - all edges besides e are nonterminal;
  - the root is the first external node and all other external nodes are leaves (where the order of the external nodes respects the order of the leaves);
  - any leaf with in-degree 1 is either external or its only incoming edge is terminal.

The figure below shows three productions in normal form.



Since RDG are a restricted form of HRG, they can also be made probabilistic. To use Theorem 6 to show that a family of languages is closed under intersection, we require knowing that the family is MSO-definable. We do not have a formal proof of that being the case for RDL (although we suspect it is). However, given two RDGs in normal form, it appears possible to construct a grammar defining their intersection. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two RDGs from which we will construct the intersection grammar $\mathcal{G}_\cap$. The set of nonterminals of $\mathcal{G}_\cap$ is the cross-product of the set of nonterminals in $\mathcal{G}_1$ and $\mathcal{G}_2$, i.e. $N_{\mathcal{G}_\cap} = \{X_1X_2 \mid X_1 \in N_{\mathcal{G}_1}, X_2 \in N_{\mathcal{G}_2}\}$. If $p_1 \in P_{\mathcal{G}_1}$ and $p_2 \in P_{\mathcal{G}_2}$ are isomorphic ignoring their nonterminal labels, then we add a production $p_\cap$ to $P_{\mathcal{G}_\cap}$ which is also isomorphic to p_1 and p_2 ignoring nonterminals. Each nonterminal edge in $p_\cap$ then has label X_1X_2 if the corresponding edges in p_1 and p_2 are labelled X_1 and X_2, respectively. Finally, we define $T_{\mathcal{G}_\cap} = T_{\mathcal{G}_1} \cap T_{\mathcal{G}_2}$ and $S_{\mathcal{G}_\cap} = S_{\mathcal{G}_1} S_{\mathcal{G}_2}$.

7.2 Tree-like grammars

To define the form of TLG, we need to first define some terms. Let R be a graph that appears on the right-hand side of a production. $\mathbb{1}^R$ is the first external node of R. Let e be an edge such that $\text{att}(e) = (v_1, v_2, v_3)$, we refer to v_1 as being the 1st node of e. The **context** of R is the set of nodes in R which are the first node of some edge in R. Recall that we use $\text{INC}_i(e, v)$ to express that v is the i-th node of e. Then the context is defined as:

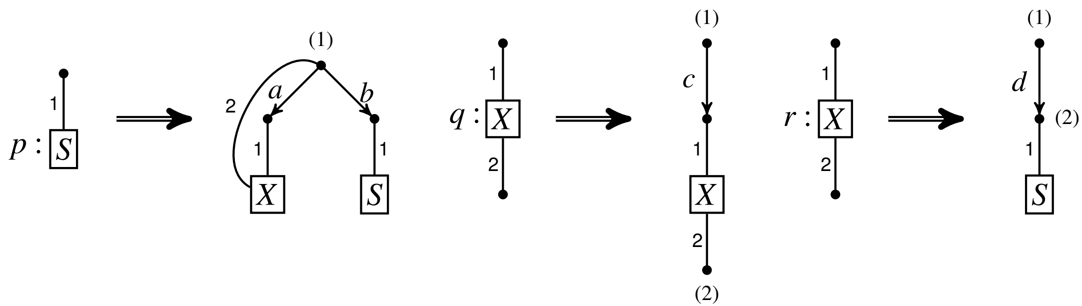
$$\text{ctxt}(R) = \{v \in V_R \mid \exists e \in E_R \text{INC}_1(e, v)\}.$$

And finally, the free nodes of R are the nodes in R which are not connected to any terminal edges:

$$\text{free}(R) = \{u \in V_R \mid u \text{ is only attached to nonterminal edges}\}$$

Definition 19. *A hypergraph R is basic tree-like if $\mathbb{1}^R \in att_R(e)$ for each terminal edge $e \in E_R$ and $ctxt(R) \cap free(R) = \emptyset$.*

Then a grammar is tree-like if each right-hand side of a production is basic tree-like and the first external nodes of two productions are never fused in a derivation. This means that we cannot have $(\mathbb{1}^{\text{RHS}(p)}, \mathbf{v}_p) \sim (\mathbb{1}^{\text{RHS}(q)}, \mathbf{v}_q)$ for any productions p and q or any derivation tree containing nodes $\mathbf{v}_p$ and $\mathbf{v}_q$. Take the productions below as an example. Each of the productions is basic tree-like. A grammar containing productions p and q would be allowed since they do not cause any first external nodes to be fused. However, a grammar containing productions p and r violates this requirement.



The tree-like languages are MSO-definable (Matheja et al., 2015) and their restriction depends on the form of the right-hand sides of the productions so Theorem 6 applies and TLL are closed under intersection. They are defined as a restriction of HRG and so they are also probabilistic.

7.3 RGL vs RDL vs TLL

It should be clear from the definitions of the formalisms that they define different grammars. For example, RDG can have at most one terminal in their productions, while RGG and TLG can have many. TLG and RDG must have all terminals connected to the first external node while RGG does not require this. On the other hand, both RDG and TLG allow nodes that are connected only to nonterminals, while RGG forbids this. Finally, RDG allows the first node of a nonterminal edge to be connected to the first external node of a clone production, but TLG never allows this.

A more interesting comparison of the formalisms is to consider the languages of graphs they can generate. An obvious distinction between the languages is that both RGG and TLG can generate cyclic graphs whereas RDG cannot. Given that semantic graph banks consist of directed acyclic graphs, we want to see if there are DAG languages that any can generate that the others cannot. We give examples of DAG languages that can be generated by some formalisms and not others.

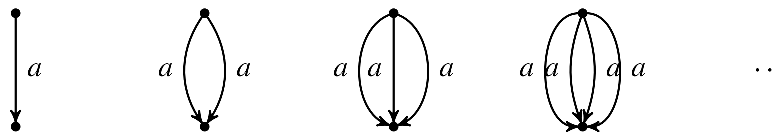
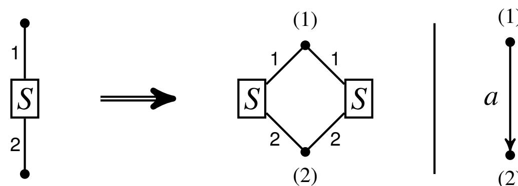


Figure 7.1: The BEACHBALL language: every graph must have exactly two nodes, and must have one or more a-edges from one to the other (always in the same direction). This language can be generated by an RDG but not by an RGG or a TLG.

Recall the BEACHBALL language from Figure 6.1 in Chapter 6, repeated here in Figure 7.1. This language can be generated by RDG but not RGG or TLG. The RDG that generates it has two productions, shown below:



This grammar is neither an RGG nor a TLG. We discussed in §6.2 why RGG cannot generate the BEACHBALL language. Tree-like grammars do not allow fusing of the first external nodes of productions. This means that, like RGG, a TLG will always

add at least one new node at each application of a production. Since the BEACHBALL language contains just two nodes in every graph, TLG cannot generate it.

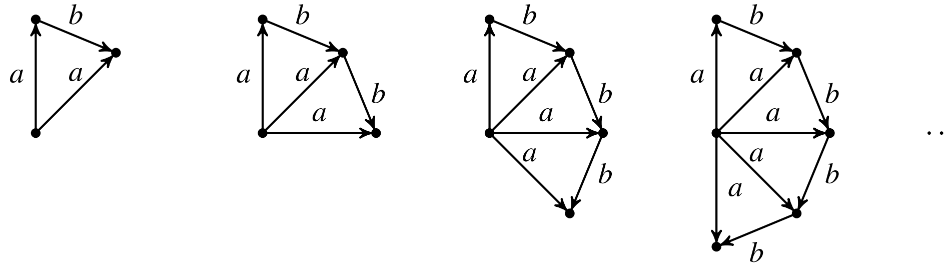
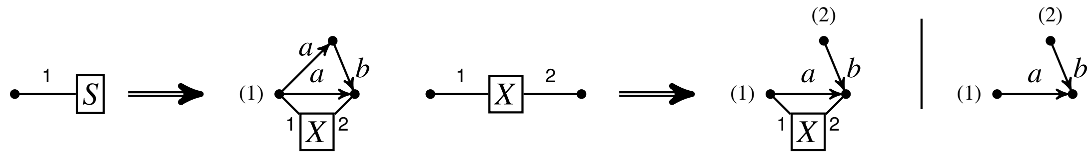
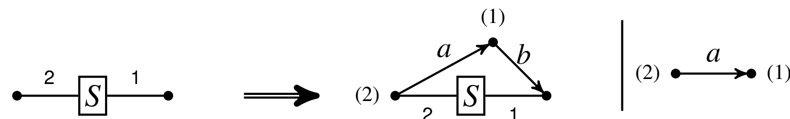


Figure 7.2: The WINDMILL language: every graph contains a single path of any length labelled by *b* edges, and one additional node. The additional node is connected to every node on the *b*-path via an *a*-edge.

The WINDMILL language (Figure 7.2) can be made by RGG and TLG but not RDG. The RGG generating the WINDMILL language is not a TLG and vice versa. The RGG is:



And the TLG is:



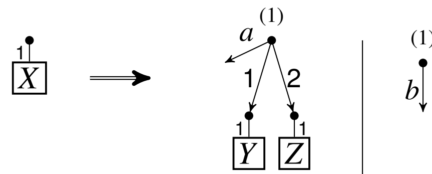
The definition of RDG means that if a node has more than one parent, then it is a leaf. Therefore, RDG cannot generate the WINDMILL language as there are nodes in the graphs that violate this requirement.[1]

- BEACHBALL $\in$ RDL$\backslash$RGL,
- BEACHBALL $\in$ RDL$\backslash$TLL,
- WINDMILL $\in$ TLL$\backslash$RDL,

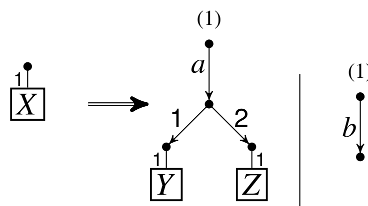
[1] In Björklund et al. (2016), they show that it is possible to interpret non-leaf nodes with more than one incoming edge as an RDL by converting each node to a hyperedge and adding extra nodes to model reentrancies.

- WINDMILL $\in$ RGL$\backslash$RDL.

We now consider the difference between TLL and RGL. TLL can generate the regular tree languages, using productions of the form:



This TLG is not an RGG since the a edge is attached to no internal nodes. As we showed in Chapter 6, RGG can simulate regular tree grammars using the form:



Since there is a one-to-one mapping between these two formats of generating regular tree languages, the difference between the structures that RGL and TLL can capture does not appear as stark as of either formalism compared with RDL. We have not found any other examples of languages that are in TLL and not RGL.

We have also not found an example of a language in RGL$\backslash$ TLL so we cannot say for certain that the three families are incomparable. Either this is the case, or RGL is a subfamily of TLL and RGL and TLL are both incomparable to RDL. We leave this as an open question.

7.4 Conclusions

While the definitions of RGG, RDG, and TLG are distinct, it is worth noting that they have some important similarities. In particular, each formalism obeys the following property: given the right-hand side of any production, the terminal subgraph is connected. This property appears to be important as it is mirrored in the restrictions of context-free string and tree languages to regular string and tree languages, respectively.

This property is used repeatedly in the proof that RGL is a subfamily of the MSO graph languages. However, both TLG and RDG allow nodes that are connected only to

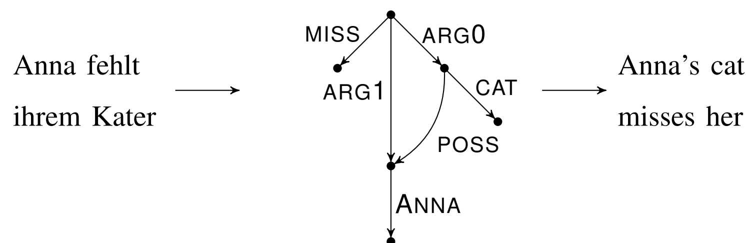
nonterminals, which is forbidden in RGGs so the proof that RGLs are MSO-definable does not apply directly to those formalisms. The study of these formalisms leaves us with a strong intuition about the forms of restrictions of HRG that generate MSO-definable languages, which we establish in the following conjecture.

Conjecture 2. *Let $\mathbb{G}$ be a restricted form of HRG such that the family of languages generated by $\mathbb{G}$ is a subfamily of SCFL. Then the terminal subgraph of the right-hand side of each production in a $\mathbb{G}$ grammar is connected.*

Chapter 8

Parsing regular graph grammars

In previous chapters, we discussed graph formalisms and their properties. This chapter moves into how we could use one such formalism (regular graph grammars) for parsing graphs. We define a graph parser based on Earley's algorithm (Earley, 1970) for parsing context-free string languages. We prove that the parser is sound and complete, and we show that the algorithm can parse regular graph languages in linear time in terms of the size of the input graph.



Recall the example above from the introduction, which we repeat here. Consider how we might use compositional semantic representations in machine translation, a two-step process in which semantic analysis is followed by generation. Jones et al. (2012) observe that this decomposition can be modelled with a pair of synchronous grammars, each defining a relation between strings and graphs. Necessarily, one projection of this synchronous grammar produces strings, while the other produces graphs, i.e., is a graph grammar. A consequence of this representation is that the complete translation process can be realised by parsing: to analyse a sentence, we parse the input string with the string-generating projection of the synchronous grammar, and read off the synchronous graph from the resulting parse. To generate a sentence, we parse the graph and read off the synchronous string from the resulting parse. In this chapter, we focus on the latter problem: using graph grammars to parse input graphs. We call

this **graph recognition** to avoid confusion with other parsing problems.

Chiang et al. (2013) work on HRG parsing and precisely characterise the complexity of a CKY-style algorithm for graph recognition from Lautemann (1990) to be polynomial in the size of the input graph. As we discussed in Chapter 5, HRGs are very expressive—they can generate graphs that simulate non-context-free string languages (Engelfriet and Heyker, 1991; Bauer and Rambow, 2016). This means they are likely more expressive than we need to represent the linguistic phenomena that appear in existing semantic datasets.

After studying RGG in detail in Chapter 6, here we aim to define a HRG parsing algorithm that is particularly efficient for RGG. We focus on RGG as a subfamily of HRG, since, like its regular counterparts among string and tree languages, it is less expressive than context-free grammars but may admit more practical algorithms. By analogy to Chiang's CKY-style algorithm for HRG, we develop an Earley-style recognition algorithm for RGLs that is linear in the size of the input graph.

The algorithm works by recognising each of the terminal edges in the right-hand side of a production in sequence. It is particularly efficient for RGL since these edges always form a connected subgraph.

The definitions of hyperedge replacement grammars and regular graph grammars can be found in Chapters 5 and 6, respectively.

Example 37. We re-use the example of an RGG from Chapter 6 and we show how it is recognised.

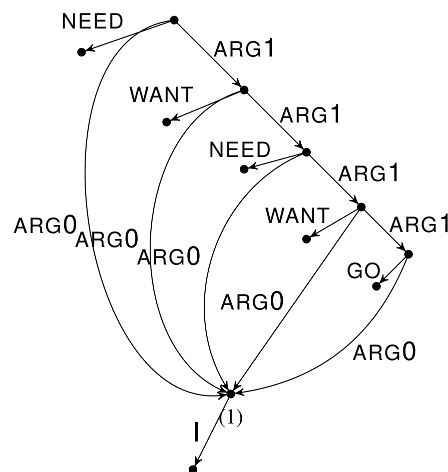


Figure 8.1: Graph derived by grammar in Table 8.1.

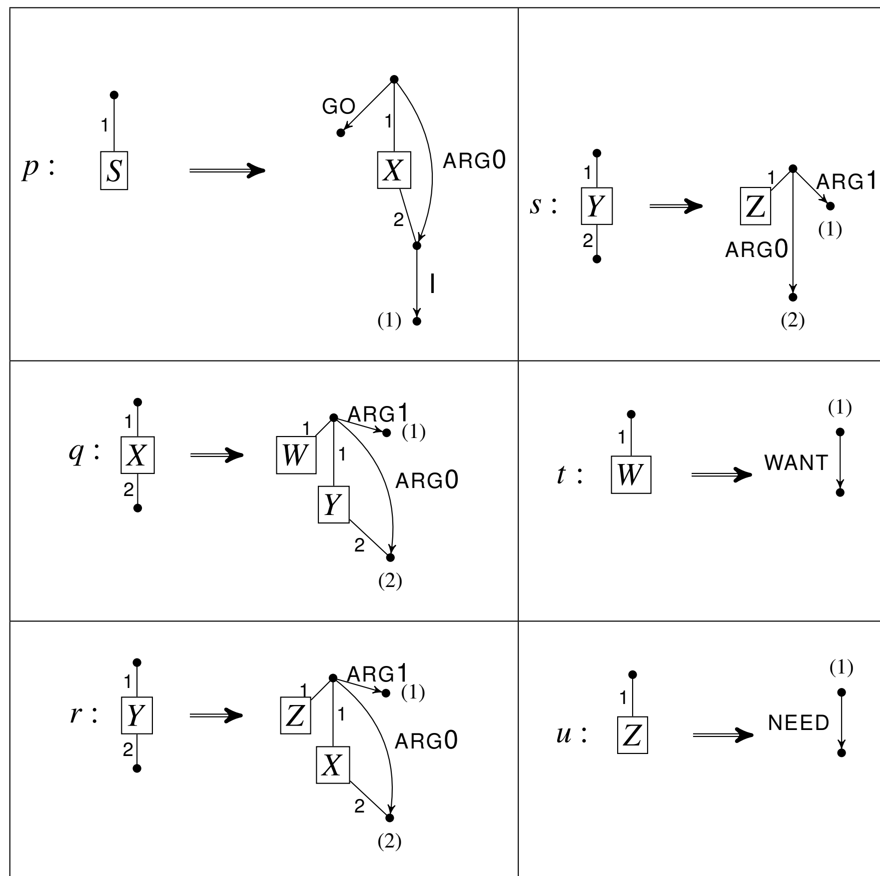


Table 8.1: Productions of an RGG (the same example as in Chapter 6. The labels p, q, r, s, t, and u label the productions so that we can refer to them in the text.

8.1 RGL recognition

To recognise RGG, we exploit the property that every nonterminal including the start symbol has rank at least one (see Definition 18 in Chapter 6), and we assume that the corresponding external node is identified in the input graph. This mild assumption may be reasonable for applications like AMR parsing, where grammars could be designed so that the external node is always the unique root. Later we relax this assumption.

The availability of an identifiable external node suggests a top-down algorithm, and we take inspiration from a top-down recognition algorithm for the predictive top-down parsable grammars, another subclass of HRG (Drewes et al., 2015). These grammars, the graph equivalent of LL(1) string grammars, are incomparable to RGG, but the algorithms are related in their use of top-down prediction and in that they both fix an order of the edges in the right-hand side of each production.

8.1.1 Top-Down recognition for RGLs

Algorithms that recognise strings using context-free string grammars work over substrings. These can be specified using the start and end indexes of the substring. The obvious generalisation of a substring to a graph is a subgraph. The way we specify a subgraph is using a **boundary representation**, which we will describe below.

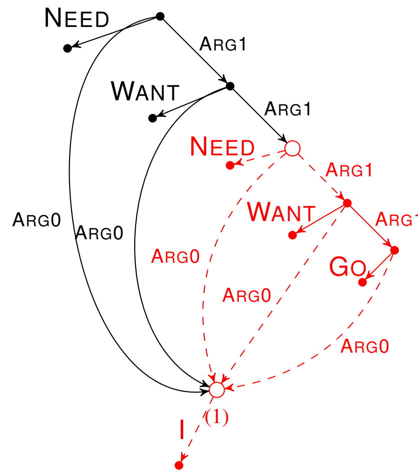
Just as the algorithm of Chiang et al. (2013) generalises CKY to HRG, our algorithm generalises Earley's algorithm (Earley, 1970). Both algorithms operate by recognising incrementally larger subgraphs of the input graph.

Definition 20. *(Chiang et al. 2013; Definition 6) Let I be a subgraph of a graph G. A **boundary node** of I is a node which is either an endpoint of an edge in $G \backslash I$ or an external node of G. A **boundary edge** of I is an edge in I which has a boundary node as an endpoint. The **boundary representation** of I is the tuple $b(I) = \langle bn(I), be(I), m \in I \rangle$ where*

1. *$bn(I)$ is the set of boundary nodes of I,*
2. *$be(I)$ is the set of boundary edges of I,*
3. *$(m \in I)$ is a flag indicating whether the marker node is in I.*

The graph below shows a subgraph in red. The boundary nodes are shown as large hollow nodes and the boundary edges are shown dashed.

Chiang et al. (2013) prove each subgraph has a unique boundary representation, and give algorithms that use only boundary representations to compute the union of



two subgraphs, requiring time linear in the number of boundary nodes; and to check disjointness of subgraphs, requiring time linear in the number of boundary edges. To show how the marker node is important to the definition, we consider an example discussed in Chiang et al. (2013). Let G be a graph and I be a subgraph. Assume $bn(I) = \emptyset$, then $be(I)$ is also empty. If $m \in I$ then $I = G$ and if $m \notin I$ then $I = \emptyset$. Therefore, just by changing the flag $m \in I$, we obtain different subgraphs.

8.2 Earley parsing

We define our graph parser as an extension of the Earley parsing algorithm (Earley, 1970) on strings. Before we go into the details of our algorithm, we first recall Earley's algorithm. The algorithm parses context-free string languages. We write the algorithm as a deductive proof system. Given a context-free grammar (N,T,P,S) and a string w, the items of the system are of the form

$$[i, X \to \alpha \bullet \beta, j],$$

where: i and j are positions in w; α and β are strings containing terminals and nonterminals; $X \to \alpha\beta$ is a production in P; and $\bullet$ indicates that we have recognised that the substring from i to j of the string can be derived by α. We add a dummy prenonterminal $S^* \notin N$ and the production $S^* \to S$.

The axiom of the system is

$$[0, S^* \to \bullet\, S, 0]$$

and from this we want to prove

$$[0, S^* \to S \bullet, |w|].$$

The symbols α, β, and γ represent strings in $(N \cup T)^*$. The system uses three types of rules depending on what symbol immediately follows the dot ($\bullet$) in the item. The first rule is SCAN which is used when the next symbol is a terminal:

$$\frac{[i, X \to \alpha \bullet a\beta, j][w_{j+1} = a]}{[i, X \to \alpha a \bullet \beta, j+1]}.$$

Essentially SCAN sees that the next terminal in the string is a and so the $\bullet$ gets moved across the a in the production $X \to \alpha a \beta$.

The second rule is PREDICT which is used when we reach a nonterminal and want to guess what that nonterminal may generate in the string at this point:

$$\frac{[i, X \to \alpha \bullet Y\beta, j][Y \to \gamma]}{[j, Y \to \bullet \gamma, j]}.$$

The point of PREDICT is that we see that there is a nonterminal Y next to be consumed and then we also see that there is a way of rewriting Y as γ and so we predict that this production has been applied at this point, creating the new item.

The final rule is COMPLETE which is used when we reach a nonterminal and we know already that that nonterminal generates a string from that point:

$$\frac{[i, X \to \alpha \bullet Y\beta, j][j, Y \to \gamma \bullet, k]}{[i, X \to \alpha Y \bullet \beta, k]}.$$

This rule allows us to move $\bullet$ past the nonterminal Y. We can only apply this rule when the endpoint of the first item (j here) matches the beginning of the next item.

Example 38. Take the context-free grammar $S \to aSb|ab$. This generates the string language $a^n b^n$. We will show how the Earley algorithm can be applied to recognise the string $aabb$. We first add in the extra nonterminal S^* and the production $S^* \to S$. The axiom is

$$[0, S^* \to \bullet S, 0],$$

and we want to prove

$$[0, S^* \to S \bullet, 4].$$

We apply PREDICT using the axiom and the production $S \to aSb$ to get:

$$\frac{[0, S^* \to \bullet S, 0][S \to aSb]}{[0, S \to \bullet aSb, 0]}. \tag{8.1}$$

We then apply SCAN on the result of Equation 8.1 and the fact that the first letter in the string is a to get:

$$\frac{[0, S \to \bullet aSb, 0][w_1 = a]}{[0, S \to a \bullet Sb, 1].} \tag{8.2}$$

We now reach a nonterminal S and so we apply PREDICT on the result of Equation 8.2 to get:

$$\frac{[0, S \to a \bullet Sb, 1][S \to ab]}{[1, S \to \bullet ab, 1]}. \tag{8.3}$$

We then apply SCAN:

$$\frac{[1, S \to \bullet ab, 1][w_2 = a]}{[1, S \to a \bullet b, 2]}$$

and SCAN again:

$$\frac{[1, S \to a \bullet b, 2][w_3 = b]}{[1, S \to ab \bullet, 3]}. \tag{8.4}$$

Now that we have reached the end of the production $S \to ab$ in the resulting item in Equation 8.4, we can apply COMPLETE on it and the item from Equation 8.3:

$$\frac{[0, S \to a \bullet Sb, 1][1, S \to ab \bullet, 3]}{[0, S \to aS \bullet b, 3]}.$$

We then apply SCAN one more time to get:

$$\frac{[0, S \to aS \bullet b, 1][w_4 = b]}{[0, S \to aSb \bullet, 4]}.$$

Finally, we complete with the axiom to reach our goal:

$$\frac{[0, S^* \to S, 0][0, S \to aSb \bullet, 4]}{[0, S^* \to S \bullet, 4]}.$$

Before we can apply this algorithm directly to graphs, we need to deal with the traversal order of the items in the productions. Within Earley parsing on strings, the order in which the characters appear in the productions is clear— we read them from left-to-right. In the context of graphs, there is not an obvious ordering of the edges. For each production p of the grammar, we choose a fixed order on the edges of $\text{RHS}(p)$, as in Drewes et al. (2015). We discuss this order in detail in §8.2.1.

As in Earley's algorithm, we use dotted productions to represent partial recognition of productions: $X \to \bar{e}_1 \dots \bar{e}_{i-1} \bullet \bar{e}_i \dots \bar{e}_n$ means that we have identified the edges $\bar{e}_1$ to $\bar{e}_{i-1}$ and that we must next recognise edge $\bar{e}_i$. We write $\bar{e}$ and $\bar{v}$ for edges and nodes in productions and e and v for edges and nodes in a derived graph, as we did in Chapter 6. When the identity of the sequence is immaterial we abbreviate it as α, for example writing $X \to \bullet\, \alpha$.

We present our recogniser as a deductive proof system (Shieber et al., 1995). The items of the recogniser are of the form

$$[b(I), p : X \to \bar{e}_1 \dots \bullet\, \bar{e}_i \dots \bar{e}_n, \phi_p],$$

Name	Rule	Conditions
PREDICT	$\dfrac{[b(I), p : X \to \bar{e}_1 \ldots \bullet \bar{e}_i \ldots \bar{e}_n, \phi_p][q : Y \to \alpha]}{[\phi_p(\bar{e}_i), q : Y \to \bullet \alpha, \phi_q^0[\text{ext}_{\text{RHS}(q)} = \phi_p(\bar{e}_i)]]}$	$\text{lab}(\bar{e}_i) = Y$
SCAN	$\dfrac{[b(I), X \to \bar{e}_1 \ldots \bullet \bar{e}_i \ldots \bar{e}_n, \phi_p][\text{att}(e) = (v_1, \ldots, v_m), \text{lab}(\bar{e}_i) = \text{lab}(e)]}{[b(I \cup \{e\}), X \to \bar{e}_1 \ldots \bullet \bar{e}_{i+1} \ldots \bar{e}_n, \phi_p[\text{att}(\bar{e}_i) = (v_1, \ldots, v_m)]]}$	$\phi_p(\bar{e}_i)(j) \in V_G \Rightarrow$ $\phi_p(\bar{e}_i)(j) = \text{vert}(e, j)$
COMPLETE	$\dfrac{[b(I), p : X \to \bar{e}_1 \ldots \bullet \bar{e}_i \ldots \bar{e}_n, \phi_p][b(J), q : Y \to \alpha \bullet, \phi_q]}{[b(I \cup J), X \to \bar{e}_1 \ldots \bullet \bar{e}_{i+1} \ldots \bar{e}_n, \phi_p[\text{att}(\bar{e}_i) = \phi_p(\text{ext}_{\text{RHS}(q)})]]}$	$\phi_p(\bar{e}_i)(j) \in V_G \Rightarrow$ $\phi_p(\bar{e}_i)(j) =$ $\phi_q(\text{ext}_{\text{RHS}(q)})(j)$, $\text{lab}(\bar{e}_i) = Y$, $E_I \cap E_J = \emptyset$

Table 8.2: The inference rules for the top-down recogniser.

where $p : X \to \bar{e}_1, \ldots, \bar{e}_n$ is a production in the grammar with the edges in order; I is a subgraph that has been recognised as matching $\bar{e}_1, \ldots, \bar{e}_{i-1}$ in p; and $\phi_p : E_{\text{RHS}(p)} \to V_G^*$ maps the endpoints of edges in $\text{RHS}(p)$ to nodes in G.

Recall that the item in Earley parsing on strings is of the form $[i, X \to \alpha \bullet \beta, j]$. It is clear to see that the middle element involving the dotted production is analogous to that on graphs. The boundary representation $b(I)$ in the graph item serves the same purpose as the pair of indices i and j in the string item, it tells us how the substructure we are currently looking at connects up to the rest of the object we are parsing. Finally, in the graph case, we need the extra function ϕ_p to keep track of how the edges interact with one another via shared nodes.

For each production p, we number the nodes in some arbitrary but fixed order. Using this, we construct the function $\phi_p^0 : E_{\text{RHS}(p)} \to V_{\text{RHS}(p)}^*$ such that for $\bar{e} \in E_{\text{RHS}(p)}$ if $\text{att}(\bar{e}) = (\bar{v}_1, \bar{v}_2)$ then $\phi_p^0(\bar{e}) = (\bar{v}_1, \bar{v}_2)$. As we match edges in the graph with edges in p, we assign the nodes $\bar{v}$ to nodes in the graph. For example, if we have an edge $\bar{e}$ in a production p such that $\text{att}(\bar{e}) = (\bar{v}_1, \bar{v}_2)$ and we find an edge e in the graph which matches $\bar{e}$, then we update ϕ_p to record this fact, written $\phi_p[\text{att}(\bar{e}) = \text{att}(e)]$. We also use ϕ_p to record assignments of external nodes. If we assign the ith external node to v, we write $\phi_p[\text{ext}_p(i) = v]$. We write ϕ_p^0 to represent a mapping with no grounded nodes.

Since our algorithm makes top-down predictions based on known external nodes, our boundary representation must cover the case where a subgraph is empty except for

these nodes. If at some point we know that our subgraph has external nodes $\phi(\bar{e})$, then we use the shorthand $\phi(\bar{e})$ rather than the full boundary representation $\langle\phi(\bar{e}), \emptyset, m \in \phi(\bar{e})\rangle$.

To keep notation uniform, we use dummy nonterminal $S^* \notin N_{\mathcal{G}}$ that derives $S_{\mathcal{G}}$ via the production p_0. For graph G, our system includes the **axiom**:

$$[\text{ext}_G, p_0 : S^* \to \bullet\ S_{\mathcal{G}}, \phi^0_{p_0}[\text{ext}_{\text{RHS}(p_0)} = \text{ext}_G]].$$

Our **goal** is to prove:

$$[b(G), p_S : S^* \to S_{\mathcal{G}}\ \bullet, \phi_{p_S}],$$

where ϕ_{p_S} has a single edge $\bar{e}$ in its domain which has label $S_{\mathcal{G}}$ in $\text{RHS}(p_S)$ and $\phi_{p_S}(\bar{e}) = \text{ext}_G$.

As in Earley's algorithm, we have three inference rules: PREDICT, SCAN and COMPLETE (Table 8.2). PREDICT is applied when the edge after the dot is nonterminal, assigning any external nodes that have been identified. SCAN is applied when the edge after the dot is terminal. Using ϕ_p, we may already know where some of the endpoints of the edge should be, so it requires the endpoints of the scanned edge to match. COMPLETE requires that each of the nodes of $\bar{e}_i$ in $\text{RHS}(p)$ have been identified, these nodes match up with the corresponding external nodes of the subgraph J, and that the subgraphs I and J are edge-disjoint.

We prove that the top-down HRG parser is sound and complete.

Proposition 6. *Let $\mathcal{G}$ be a HRG, let G be a graph, and let $S = S_{\mathcal{G}}$. Then $G \in \mathcal{L}(\mathcal{G})$ if and only if the goal $[b(G), p_S : S^* \to S\ \bullet, \phi_{p_S}]$ can be proved from the axiom $[ext_G, p_S : S^* \to \bullet\ S, \phi^0_{p_S}[ext_{p_S} = ext_G]]$.*

Proof. For each nonterminal $X \in N_{\mathcal{G}}$, we add in a **pre-nonterminal** X^* and a production $p_X : X^* \to X$. We will prove that for each nonterminal X, $G \in \mathcal{L}_X(\mathcal{G})$ if and only if $[b(G), p_X : X^* \to X\ \bullet, \phi_{p_X}]$ can be proved from the axiom $[\text{ext}_G, p_X : X^* \to \bullet\ X, \phi^0_{p_X}[\text{ext}_{\text{RHS}(p_X)} = \text{ext}_G]]$. Greek letters α, β, γ and δ will be used throughout the proof to refer to an ordered sequence of edges in a production. To save space in the equations, we will use $\text{AXIOM}(X, G)$ to refer to the item $[\text{ext}_G, p_X : X^* \to \bullet\ X, \phi^0_{p_X}[\text{ext}_{\text{RHS}(p_X)} = \text{ext}_G]]$ and $\text{GOAL}(X, G)$ to refer to $[b(G), p_X : X^* \to X\ \bullet, \phi_{p_X}]$.

We prove by induction on the number of edges in G.

Base Case: In the base case, G consists of a single edge. We assume that every production has at least one terminal edge.

If: Assume that $\text{GOAL}(X,G)$ can be proved from $\text{AXIOM}(X,G)$. Then a COMPLETE rule of the form

$$\frac{\text{AXIOM}(X,G)[b(G),q:X\rightarrow\alpha\bullet,\phi_q]}{\text{GOAL}(X,G)}$$

must have been applied. We know that G consists of a single edge and that each production has at least one terminal edge, so it must be that α is just the single edge in G. Therefore, there is a production $q:X\rightarrow G\in P_{\mathcal{G}}$ and so $G\in\mathcal{L}_X(\mathcal{G})$.

Only If: Assume that $G\in\mathcal{L}_X(\mathcal{G})$. Since G only has one edge, there must be some production $q:X\rightarrow G\in P_{\mathcal{G}}$. Let the only edge of G be e and let the only edge of $\text{RHS}(q)$ be $\bar{e}$. Starting from $\text{AXIOM}(X,G)$, we apply PREDICT:

$$\frac{\text{AXIOM}(X,G)[q:X\rightarrow\bar{e}]}{[\phi_{p_X}(X),q:X\rightarrow\bullet\,\bar{e},\phi_q^0[\text{ext}_{\text{RHS}(q)}=\phi_{p_X}(X)]]}.$$

We can then apply SCAN to get:

$$\frac{[\phi_{p_X}(X),X\rightarrow\bullet\,\bar{e},\phi_q^0[\text{ext}_{\text{RHS}(q)}=\phi_{p_X}(X)]][\text{att}(\bar{e})\subseteq\text{att}(e)]}{[b(e),q:X\rightarrow\bar{e}\bullet,\phi_q[\text{att}(\bar{e})=\text{att}(e)]]},$$

where $\text{att}(\bar{e})\subseteq\text{att}(e)$ is shorthand for the SCAN condition.

And we apply COMPLETE to get:

$$\frac{\text{AXIOM}(X,G)[b(e),q:X\rightarrow\bar{e}\bullet,\phi_q[\text{att}(\bar{e})=\text{att}(e)]]}{[b(e),p_X:X^*\rightarrow X\bullet,\phi_{p_X}[\text{att}(X)=\phi_q(\text{ext}_{\text{RHS}(q)})]]}.$$

Since $e=G$, we have proved $\text{GOAL}(X,G)$ from $\text{AXIOM}(X,G)$.

Therefore, when G consists of a single edge, $G\in\mathcal{L}_X(\mathcal{G})$ if and only if $\text{GOAL}(X,G)$ can be proved from $\text{AXIOM}(X,G)$.

Assumption: Assume that if G has fewer than k edges then $G\in\mathcal{L}_X(\mathcal{G})$ if and only if $\text{GOAL}(X,G)$ can be proved from $\text{AXIOM}(X,G)$.

Inductive Case: Let G be a graph with k edges.

If: Assume that $\text{GOAL}(X,G)$ can be proved from $\text{AXIOM}(X,G)$. Then a COMPLETE inference rule of the form

$$\frac{\text{AXIOM}(X,G)[\text{b}(G),q:X\rightarrow\alpha\bullet,\phi_q]}{\text{GOAL}(X,G)}$$

must have been applied. Let $Y_1,\dots,Y_n$ be the nonterminal edges in α and let $a_1,\dots,a_m$ be the terminal edges in α. For the item $\text{GOAL}(X,G)$ to be proved, we must have applied SCAN on each a_i for $i\in[m]$. We also must have applied COMPLETE rules of the form:

$$\frac{[\text{b}(G_j),q:X\rightarrow\beta\bullet Y_j\gamma,\phi_q][\text{b}(H_j),r_j:Y_j\rightarrow\delta_j\bullet,\phi_{r_j}]}{[\text{b}(G_j\cup H_j),q:X\rightarrow\beta Y_j\bullet\gamma,\phi_q\cup\phi_{r_j}]}$$

such that for each $j\in[n]$ G_j is the subgraph of G that is recognised before we PREDICT Y_j, and G is made up of the terminal graph formed by the set of terminal edges a_i for

$i \in [m]$ and the subgraphs H_j for $j \in [n]$. For each $j \in [n]$, we can apply COMPLETE to get that:

$$\frac{\text{AXIOM}(Y_j,H_j)[b(H_j),r_j : Y_j \to \delta_j \bullet, \phi_{r_j}]}{\text{GOAL}(Y_j,H_j)}.$$

The number of edges in each H_j is less than k and so by the inductive hypothesis, $H_j \in \mathcal{L}_{Y_i}(\mathcal{G})$ for each $j \in [n]$. The production $q : X \to \alpha$ contains nonterminals $Y_1,\ldots,Y_n$ and $G = \alpha[Y_1/H_1]\ldots[Y_n/H_n]$, therefore $G \in \mathcal{L}_X(\mathcal{G})$.

Only If: Assume that $G \in \mathcal{L}_X(\mathcal{G})$. Then there exists some production $X \to H$ such that H has nonterminals $Y_1,\ldots,Y_n$ and there exist graphs $H_1,\ldots,H_n$ such that $H_i \in \mathcal{L}_{Y_i}(\mathcal{G})$ for each $i \in [n]$ and $G = H[Y_1/H_1]\ldots[Y_n/H_n]$. Since H must contain at least one terminal edge, each H_i contains fewer than k edges. By the inductive hypothesis, GOAL(Y_i,H_i) can be proved from AXIOM(Y_i,H_i) for each $i \in [n]$. Therefore, for each $i \in [n]$, a COMPLETE rule of the form:

$$\frac{\text{AXIOM}(Y_i,H_i)[\text{b}(H_i),r_i : Y_i \to \beta_i \bullet, \phi_{r_i}]}{\text{GOAL}(Y_i,H_i)}$$

must have been applied.

Let $a_1,\ldots,a_m$ be the terminal edges in H and let α be some fixed ordering of the set $\{Y_1,\ldots,Y_n\} \cup \{a_1,\ldots,a_m\}$. Using AXIOM$(X,G)$, we can use PREDICT to prove:

$$\frac{\text{AXIOM}(X,G)[q : X \to \alpha]}{[\phi_{p_X}(X), X \to \bullet\, \alpha, \phi^0_q[\text{ext}_{\text{RHS}(q)} = \phi_{p_X}(X)]]}.$$

And then for $i \in [n+m]$ if α_i is a terminal we apply SCAN:

$$\frac{[\text{b}(G_i), q : X \to \beta \bullet \alpha_i\gamma, \phi_q][\text{att}(e) = (v_1,\ldots,v_m), \text{lab}(e) = \text{lab}(\alpha_i)]}{[[\text{b}(G_i \cup \{e\}), X \to \beta\alpha_i \bullet \gamma, \phi_q[\text{att}(\alpha_i) = (v_1,\ldots,v_m)]]}$$

where G_i is the subgraph recognised up until we SCAN α_i, β is the sequence of edges from α_1 to α_{i-1} and γ is the sequence from α_{i+1} to α_{n+m}. If α_i is nonterminal (say Y_i) we apply COMPLETE:

$$\frac{[\text{b}(G_i), q : X \to \beta \bullet \alpha_i\gamma, \phi_q][[\text{b}(H_i), r_i : Y_i \to \beta_i \bullet, \phi_{r_i}]}{[\text{b}(G_i \cup H_i), X \to \beta\alpha_i \bullet \gamma, \phi_q \cup \phi_{r_i}]}$$

where β is the sequence α_1 to α_{i-1} and γ is the sequence α_{i+1} to α_{n+m}. Using a combination of SCAN and COMPLETE in this fashion, we will eventually reach the item $[b(G), q : X \to \alpha \bullet, \phi_q]$ and so we can apply COMPLETE a final time to produce:

$$\frac{\text{AXIOM}(X,G)[b(G), q : X \to \alpha \bullet, \phi_q]}{\text{GOAL(X,G)}}.$$

Therefore, for graphs of any size, $G \in \mathcal{L}_X(\mathcal{G})$ if and only if GOAL(X,G) can be proved from AXIOM(X,G). □

Example 39. Using the RGG in Table 8.1, we show how to recognise the graph in Figure 8.2, which can be derived by applying production s followed by production

u, where the external nodes of Y are (v_3, v_2). Assume the ordering of the edges in production s is ARG1, ARG0, Z; the top node is $\bar{v}_1$; the bottom node is $\bar{v}_2$; and the node on the right is $\bar{v}_3$; and that the marker node is not in this subgraph—we elide reference to it for simplicity. Let $\bar{v}_4$ be the top node of $\text{RHS}(u)$ and $\bar{v}_5$ be the bottom node of $\text{RHS}(u)$. The external nodes of Y are determined top-down, so the recognition of this subgraph is triggered by this item:

$$[\{v_3, v_2\}, Y \rightarrow \bullet \text{ ARG1 ARG0} Z, \phi_s^0[\text{ext}_{\text{RHS}(s)} = (v_3, v_2)]], \tag{8.5}$$

where $\phi_s(\text{ARG1}) = (\bar{v}_1, v_3)$, $\phi_s(\text{ARG0}) = (\bar{v}_1, v_2)$, and $\phi_s(Z) = (\bar{v}_1)$.

Table 8.3 shows how we can prove the item

$$[\langle\{v_3, v_2\}, \{e_3, e_2\}\rangle, Y \rightarrow \text{ARG1ARG0}Z \bullet, \phi]$$

The boundary representation $\langle\{v_3, v_2\}, \{e_3, e_2\}\rangle$ in this item represents the whole subgraph shown in Figure 8.2.

Current Item	Reason
1. $[\{v_3, v_2\}, Y \rightarrow \bullet \text{ ARG1ARG0}Z, \phi_s^0[\text{ext}_{\text{RHS}(s)} = (v_3, v_2)]]$	Equation 8.5
2. $[\langle\{v_3, v_2, v_1\}, \{e_3\}\rangle, Y \rightarrow \text{ARG1} \bullet \text{ARG0}Z, \phi_s[\text{att}(\text{ARG1}) = (v_1, v_3)]]$	SCAN: 1., $\text{att}(e_3) = (v_1, v_3)$, $\text{lab}(e_3) = \text{ARG1}$
3. $[\langle\{v_3, v_2, v_1\}, \{e_3, e_2\}\rangle, Y \rightarrow \text{ARG1ARG0} \bullet Z, \phi_s[\text{att}(\text{ARG0}) = (v_1, v_2)]]$	SCAN: 2., $e_2 = \text{att}(v_1, v_2)$, $\text{lab}(e_2) = \text{ARG0}]$
4. $[(v_1), Z \rightarrow \bullet \text{ NEED}, \phi_u^0[\text{ext}_{\text{RHS}(u)} = (v_1)]]$	PREDICT: 3. and $Z \rightarrow \text{NEED}$
5. $[\langle\{v_1, v_4\}, \{e_1\}\rangle, Z \rightarrow \text{NEED} \bullet, \phi_u[\text{att}(\text{NEED}) = (v_1, v_4)]]$	SCAN: 4., $\text{att}(e_1) = (v_1, v_4)$, $\text{lab}(e_1) = \text{NEED}$
6. $[\langle\{v_3, v_2\}, \{e_3, e_2\}\rangle, Y \rightarrow \text{ARG1ARG0}Z \bullet, \phi_s[\text{att}(Z) = (v_1)]]$	COMPLETE: 3. and 5.

Table 8.3: The steps of recognising that the subgraph shown in Figure 8.2 is derived from productions r_2 and u in the grammar in Table 8.1.

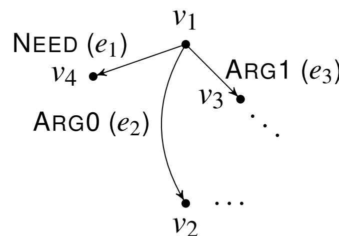


Figure 8.2: Top left subgraph of Figure 8.1. To refer to nodes and edges in the text, they are labeled v_1, v_2, v_3, e_1, e_2, and e_3.

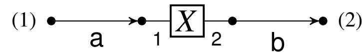


Figure 8.3: The edges of this graph have no connected ordering.

8.2.1 Connected ordering

Our algorithm requires a fixed ordering of the edges in the right-hand side of each production. In general, the algorithm works for any HRG and using any ordering of the edges. In the case of RGG, we can constrain the ordering which allows us to bound the recognition complexity. If $s = \bar{e}_1 \dots \bar{e}_n$ is an order, define $s_{i:j} = \bar{e}_i \dots \bar{e}_j$.

Definition 21. *Let $s = \bar{e}_1, \dots, \bar{e}_n$ be an edge order of a right-hand side of a production. Then s is* ***connected*** *if it has the following properties:*

1. *$\bar{e}_1$ is connected to an external node,*
2. *$s_{1:j}$ is a connected graph for all $j \in [n]$*
3. *if $\bar{e}_i$ is nonterminal, each endpoint of $\bar{e}_i$ must be incident to some terminal edge $\bar{e}_j$ for which $j < i$.*

Example 40. The ordering of the edges of production s in Example 39 is connected.

Arbitrary HRGs do not necessarily admit a connected ordering. For example, the graph in Figure 8.3 cannot satisfy Properties 2 and 3 simultaneously. However, RGGs do admit a connected ordering.

Proposition 7. *If $\mathcal{G}$ is an RGG, for every $p \in P_{\mathcal{G}}$, there is a connected ordering of the edges in $\text{RHS}(p)$.*

Proof. If $\text{RHS}(p)$ contains a single node then it must be an external node and it must have a terminal edge attached to it since $\text{RHS}(p)$ must contain at least one terminal edge. If $\text{RHS}(p)$ contains multiple nodes then by C2 of the definition of RGG, there must be terminal internal paths between all of them, so there must be a terminal edge attached to the external node, which we use to satisfy Property 1. To produce a connected ordering, we next select terminal edges once one of their endpoints is connected to an ordered edge, and nonterminal edges once all endpoints are connected to ordered edges, possible by C2. Therefore, Properties 2 and 3 are satisfied. □

A connected ordering tightly constrains the recognition of edges. Property 3 ensures that when we apply PREDICT, the external nodes of the predicted edge are all bound to specific nodes in the graph. Properties 1 and 2 ensure that when we apply SCAN, at least one endpoint of the edge is bound (fixed).

8.2.2 Recognition complexity

Assume a connected-ordered RGG. Let the maximum number of edges in the right-hand side of any production be m; the maximum number of nodes in any right-hand side of a production k; the maximum degree of any node in the input graph d; and the number of nodes in the input graph n.

Drewes et al. (2015) also propose a HRG recogniser which can recognise a subclass of HRG (incomparable to RGG) called the predictive top-down parsable grammars. Their recogniser, in this case, runs in $O(n^2)$ time. A well-known bottom-up recognising algorithm for HRG was first proposed by Lautemann (1990), where the recogniser is shown to be polynomial in the size of the input graph. Later, Chiang et al. (2013) formulate the same algorithm more precisely and show that the recognising complexity is $O((3^d \times n)^{k+1})$ where k in their case is the treewidth of the grammar.[1]

Remark 3. *The maximum number of nodes in any right-hand side of a production (k) is also the maximum number of boundary nodes for any subgraph in the recogniser.*

COMPLETE combines subgraphs I and J only when the entire subgraph derived from Y has been recognised. Boundary nodes of J are also boundary nodes of I because they are nodes in the terminal subgraph of $\text{RHS}(p)$ where Y connects. The boundary nodes of $I \cup J$ are also bounded by k since form a subset of the boundary nodes of I.

Remark 4. *Given a boundary node, there are at most $(d^m)^{k-1}$ ways of identifying the remaining boundary nodes of a subgraph that is isomorphic to the terminal subgraph of the right-hand side of a production.*

The terminal subgraph of each production is connected, by C3 of the definition of RGG, with a maximum path length of m. For each edge in the path, there are at most d subsequent edges. Hence for the $k-1$ remaining boundary nodes, there are $(d^m)^{k-1}$ ways of choosing them. If we did not require the terminal subgraph to be connected this term would depend on n instead of d. In that case, we would have to look through the entire graph for these nodes, not just those connected via a sequence of edges of length at most m. Similarly, if we did not follow a connected ordering of the edges then we could end up looking for some node anywhere in the graph, adding a factor of n rather than following from terminal edges we have already recognised.

[1] Informally, the treewidth of a graph is a measure of how similar it is to a tree. To give an intuition, the treewidth of a tree is 1 while the treewidth of a complete graph with n nodes is n.

We count instantiations of COMPLETE to get an upper bound on the complexity (McAllester, 2002), using similar logic to Chiang et al. (2013). The number of boundary nodes of I, J and $I \cup J$ is at most k. Therefore, if we choose an arbitrary node to be some boundary node of $I \cup J$, there are at most $(d^m)^{k-1}$ ways of choosing its remaining boundary nodes. For each of these nodes, there are at most $(3^d)^k$ states of their attached boundary edges: in I, in J, or in neither. The total number of instantiations is $O(n(d^m)^{k-1}(3^d)^k)$, linear in the number of input nodes and exponential in the degree of the input graph. Note that in the case of the AMR dataset (Banarescu et al., 2013), the maximum node degree is 17 and the average is 2.12 (Chiang et al., 2018). Note that d depends on the input graph but k and m by the grammar. To minimise k and m, it could be possible to break the productions in the grammar down into smaller pieces (similar to the tree decomposition used in Chiang et al. (2013)). However, we would need to be careful to preserve the form of the productions so that each still admits a connected ordering.

We observe that RGGs could be relaxed to produce graphs with no external nodes by adding a dummy nonterminal S' with rank 0 and a single production $S' \to S$. To adapt the recognition algorithm, we would first need to guess where the graph starts. This would add a factor of n to the complexity as the graph could start at any node.

8.3 Conclusions

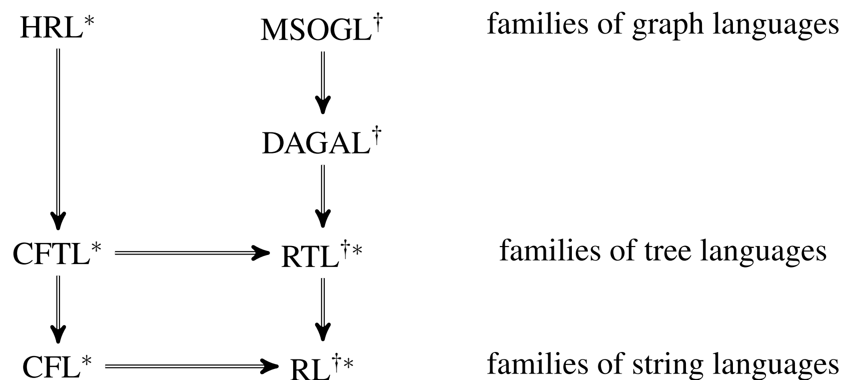
We presented an Earley-style recognition algorithm for hyperedge replacement grammars. We have shown that the algorithm is particularly efficient in recognising regular graph grammars. As mentioned in Chapter 6, RGG may be too restrictive to model meaning representations and so we leave it as an open question whether there is a more general formalism which is more suited and also admits efficient parsing algorithms.

In Chapter 7, we discussed the differences between RGG, RDG and TLG. Since RDG and TLG are both defined as restricted forms of HRG, the algorithm presented in this chapter can also be used to recognise them. The efficiency gained by using the connected ordering defined here does not immediately apply to RDG and TLG since both formalisms allow nodes in productions that are only connected to nonterminal edges—something which RGG forbids. Given the similarities between RGG, RDG and TLG, we believe it is likely that this algorithm would be efficient on the other two formalisms but we leave a precise analysis of the complexity as an open problem.

Chapter 9

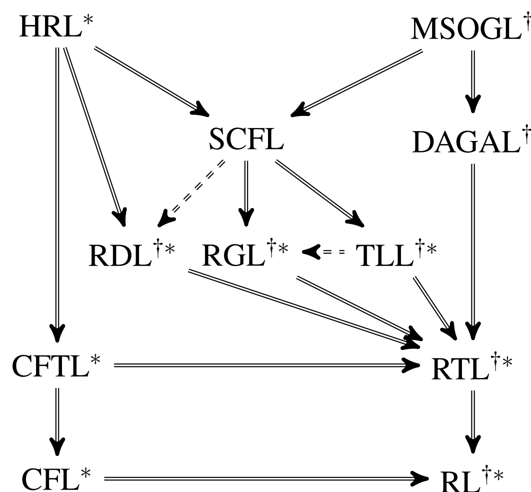
Conclusions

In the introduction to this thesis, we expressed a desire to have a model of graphs that has similar properties to finite-state models of strings and trees. In particular, we sought a model that (1) could be made probabilistic; and (2) whose graph languages are closed under intersection. We studied two graph formalisms which have been considered for NLP applications in the past: the DAG automata languages (DAGAL), and the hyperedge replacement languages (HRL). We then also studied the monadic second-order graph languages (MSOGL), which have not been previously studied in an NLP context. The Hasse diagram below summarises what we knew about the relationships between these families at the beginning of this thesis. It also shows the properties of the families—a $*$ indicates the family has a probabilistic extension and a † indicates that the languages are closed under intersection.



In this thesis, we have filled in some of the gaps in this space. In terms of the previous diagram, we now know that DAGAL are not probabilistic (or at least not by weighting transitions as is done in finite-state automata on strings and trees). The figure below shows some of what we know now. Possible relationships are depicted using dashed

edges. The strongly context-free languages (SCFL) are defined (non-constructively) as the family of languages which are both in HRL and MSOGL (Courcelle, 1991). Within SCFL, there are several families defined by restrictions on hyperedge replacement grammars that we considered. These include the regular graph languages (RGL) and the tree-like languages (TLL). We also considered the restricted DAG languages (RDL) which we believe is likely to be a subfamily of SCFL as well, but this has not been formally proven. In Chapter 7, we compared these three families in terms of their expressivity and found that RGL and RDL are incomparable; RDL and TLL are incomparable; and RGL are either a subfamily of TLL, or are incomparable to TLL.



9.1 Open theoretical problems

From the above diagram, it should be clear that there is much work left to be done in this space. The following is a (certainly incomplete) list of some open problems and questions:

- Is there a subclass of DAG automata that are probabilistic? We discussed this in §3.5.
- Is there another way of assigning weights to DAGs using DAG automata that defines a probability distribution? We also discussed this in §3.5.
- Is there a probabilistic extension of MSO on graphs? A good starting point would be to study the automaton described in Reiter (2014) which defines the MSO graph languages.

- Is there a formalism defining the strongly context-free languages? We believe that intuitions from RGL, RDL and TLL may help here (see Conjecture 2).
- Are RGL contained within TLL or are the families incomparable?
- Is RDL a subfamily of MSOGL (and therefore SCFL)? We believe this is likely due to the fact that the derivation structure of RDL graphs can be easily extracted.
- What is the complexity of parsing RDL and TLL using the algorithm described in Chapter 8?

9.2 Graph formalisms in practice

Concurrently with the work in this thesis, there has been a lot of work on semantic parsing with neural networks (i.e. continuous rather than finite-state models). Lyu and Titov (2018) describe an abstract meaning representation (AMR) parser which uses an Erdös Renyii (Erdős and Rényi, 1959) style of generating graphs. In this setup, the decoder first generates a sequence of nodes and then generates the edges connecting those nodes. Buys and van der Merwe (2013) propose a transition based parser for minimal recursion semantics which also works well on AMR. Li et al. (2018) propose a model for generating graphs which involves generating nodes sequentially. Each time a new node is generated, the model decides whether to attach this new node to any of the previously generated nodes. These models are very flexible in the sets of graphs that they can generate—the space of graphs is not defined by some grammar or automaton. A major difficulty with this type of approach is that graphs are inherently unordered but recurrent neural network models generate sequences. This means that either some order over the graphs has to be decided, or the traversal of the graph is modelled by a latent variable.

A natural question at this point is whether any of the work described in this thesis is relevant to neural network models for meaning representations. We believe that it is, and that we can combine grammar formalisms with neural network models effectively. Recently, Groschwitz et al. (2018) used the HR algebra to parse AMR graphs. The graph languages generated by the HR algebra are the same as those generated by HRG (Courcelle and Engelfriet, 2011), but they generate graphs by composing subgraphs by fusing specially marked nodes rather than carrying out nonterminal replacement. Intuitively, HRG generate graphs top-down while the HR algebra generates graphs bottom-up.

Their paper works by tagging each word in the input sentence with a (possibly empty) graph fragment. They then use a neural dependency parser to predict a dependency tree over these fragments. The dependency tree is then deterministically converted into a derivation tree for the algebra, and the derivation tree tells them how to piece the fragments together to generate the final graph. A major difficulty with this approach is how to generate the set of fragments from the training data graphs. The HR algebra (as well as HRG) is very flexible and so there are many ways of extracting a derivation tree and set of graph fragments from a graph. For that reason, Groschwitz et al. (2018) use a restricted form of the HR algebra, called the AM algebra, which produces far fewer derivation trees per graph. The parsing results in this paper at the time of publication were state-of-the-art, showing that using graph structures in NLP graph applications can really work.

We believe that the restricted DAG grammars (RDG) may be useful for semantic parsing. As the name suggests, they are very restricted in the DAGs they can generate but Björklund et al. (2016) show how AMR graphs can be adapted so that they can be parsed by RDG. Part of this adaptation involves choosing a traversal order of the graph, one reasonable order could be the order in which the AMR is annotated in the data. Given a set of such graphs, we can use RDG to extract a grammar and a set of derivation trees such that each graph has exactly one derivation tree. Similar to Groschwitz et al. (2018), we then would propose to use a neural network model to learn the mapping from sentences to these derivation trees rather than the graphs themselves. An attractive aspect of this approach is that it can be used for any graph bank—the grammar extraction is agnostic to the annotation scheme of the data. This is in contrast to Groschwitz et al. (2018) where the grammar extraction is tailored specifically to the AMR data.

We hope that this thesis can serve as a more accessible introduction to a very formal field. We have highlighted many open theoretical and practical problems which we look forward to seeing tackled in the coming years.

Appendix A

Extra proofs from Chapter 6

Lemma 7 (Lemma 5.7 of Courcelle (1991))**.** Let $\mathcal{G}$ be an RGG, G be in $\mathcal{L}(\mathcal{G})$, $\mathbf{T}$ be a derivation tree of G, and α be the parameter assignment defined with respect to $\mathbf{T}$. One can construct a formula $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$ such that, for $u,w \in V_G \cup E_G$:

$$(G,u,w,\alpha) \models \text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$$

iff $u = h(\bar{c}_p,\mathbf{v}), w = h(\bar{c}_{p'},\mathbf{v}')$ for some $\mathbf{v},\mathbf{v}'$ in $V_\mathbf{T}$ where $p = lab_\mathbf{T}(\mathbf{v})$, $p' = lab_\mathbf{T}(\mathbf{v}')$, $\mathbf{v}$ is an i-child, and $\mathbf{v}'$ is the i'-th child of $\mathbf{v}$ in $\mathbf{T}$.

Proof. For every p, p' in P, every $i \in [0,|\text{NT}(P)|]$, every $i' \in [|\text{NT}(p)|]$, we construct a formula $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$. Let $\bar{x}_1,\ldots,\bar{x}_k$ be the sequence of nodes of the i'-th nonterminal edge of $\text{RHS}(p)$. We let $(\bar{y}_1,\ldots,\bar{y}_k)$ be the sequence of external nodes of $\text{RHS}(p')$. Define:

$$\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W}):$$
$$\exists v_1,\ldots,v_k\Big(\text{ANC}_{p,i,\bar{x}_1}(u,v_1,\mathcal{W}) \wedge \text{ANC}_{p',i',\bar{y}_1}(w,v_1,\mathcal{W}) \wedge \cdots \wedge \text{ANC}_{p,i,\bar{x}_k}(u,v_k,\mathcal{W}) \wedge \text{ANC}_{p',i',\bar{y}_k}(w,v_k,\mathcal{W})\Big).$$

If:

Let $u = h_v(\bar{c}_p,\mathbf{v}), w = h_v(\bar{c}_{p'},\mathbf{v}')$ for some $\mathbf{v},\mathbf{v}'$ such that $\text{lab}_\mathbf{T}(\mathbf{v}) = p$, $\text{lab}_\mathbf{T}(\mathbf{v}') = p'$, and $\mathbf{v}'$ is the i'-th child of $\mathbf{v}$ in $\mathbf{T}$. Then for each $j \in [k]$ there is some $v_j = h_v(\bar{x}_j,\mathbf{v}) = h_v(\bar{y}_j,\mathbf{v}')$ by the definition of HRG—these are the nodes fused together as replacement is carried out. It follows from Lemma 5 that $\text{ANC}_{p,i,\bar{x}_j}(u,v_j,\mathcal{W})$ and $\text{ANC}_{p',i',\bar{y}_j}(w,v_j,\mathcal{W})$ hold for all $j \in [k]$. Hence, $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$ holds.

Only if:

Conversely, assume that $\text{PAR}_{p,i,p',i'}(u,w,\mathcal{W})$ holds. We want to show that $u = h(\bar{c}_p,\mathbf{v})$, $w = h(\bar{c}_{p'},\mathbf{v}')$, $\text{lab}_\mathbf{T}(\mathbf{v}) = p$, $\text{lab}_\mathbf{T}(\mathbf{v}') = p'$, and $\mathbf{v}'$ is the i-th child of $\mathbf{v}$. Since PAR

holds, we know that there exist nodes $v_1,\ldots,v_k$ such that $\text{ANC}_{p,i,\bar{x}_j}(u,v_j,\mathcal{W})$ and $\text{ANC}_{p',i',\bar{y}_j}(w,v_j,\mathcal{W})$ hold for all j. The fact that these ANC formulas hold mean that $u = h_v(\bar{c}_p,\mathbf{v})$ and $w = h_v(\bar{c}_{p'},\mathbf{v}')$ for some node $\mathbf{v}$ labelled p which is an i-child and $\mathbf{v}'$ labelled p' which is an i'-child. It remains to show that $\mathbf{v}'$ is the i'-th child of $\mathbf{v}$.

Since PAR holds, $h_v(\bar{x}_j,\mathbf{v}) = h_v(\bar{y}_j,\mathbf{v}')$ for all j. By C2 of RGG, some node $\bar{x}_j$ must be internal in $\text{RHS}(p)$. It follows from Lemma 2 that $\mathbf{v}$ is an ancestor of $\mathbf{v}'$. If $\mathbf{v}'$ is not a child of $\mathbf{v}$ then $\mathbf{v}'$ is the child of some node $\mathbf{v}'' \neq \mathbf{v}$ and $\mathbf{v}$ is an ancestor of $\mathbf{v}''$. Since $\mathbf{v}''$ is the parent of $\mathbf{v}'$ and has $\mathbf{v}$ as an ancestor, by Lemma 1, there must be a set of nodes $\{\bar{z}_1,\ldots,\bar{z}_k\}$ in $\text{RHS}(\text{lab}(\mathbf{v}''))$ such that $v_j = h_v(\bar{z}_j,\mathbf{v}'')$ for each $j \in [k]$. Since $\bar{y}_1,\ldots,\bar{y}_k$ are the external nodes of $\text{RHS}(p')$, the set of nodes $\bar{z}_1,\ldots,\bar{z}_k$ must be the set of nodes of the nonterminal $\text{LHS}(p')$ in $\text{RHS}(p'')$. But, by C2 of RGG, there must be some $j \in [k]$ such that $\bar{z}_i$ is internal. Then, $v_j = h_v(\bar{x}_j,\mathbf{v}) = h_v(\bar{z}_j,\mathbf{v}'')$ and $\bar{z}_j$ is internal and so $\mathbf{v}''$ is an ancestor of $\mathbf{v}$. This is a contradiction since we already assumed that $\mathbf{v}$ was an ancestor of $\mathbf{v}''$. Therefore, $\mathbf{v}'$ is the i'-th child of $\mathbf{v}$. □

To avoid needing to switch between the appendix and the main text, we repeat the formula SUBGRAPH($\mathcal{W}$) here which is defined in terms of S1,S2 and S3.

$$\text{S1} : \forall v \exists e \vee_j \text{INC}_j(e,v).$$

We define S2 and S3 in terms of a specific i, p, and j:

$$\text{S2}_{i,p,j}(\mathcal{W}) : \forall c \in C_{i,p} \exists! e \in E_{i,p,j}$$
$$\exists v_1 \text{ANC}_{p,i,\bar{x}_1}(c,v_1,\mathcal{W}) \wedge \cdots \wedge \exists v_{j_k} \text{ANC}_{p,i,\bar{x}_{j_k}}(c,v_{j_k},\mathcal{W}) \wedge \text{edge}^2(e,v_1,\ldots,v_{j_k});$$

and

$$\text{S3}_{i,p,j}(\mathcal{W}) : \forall e \in E_{i,p,j} \exists! c \in C_{i,p}$$
$$\exists v_1 \text{ANC}_{p,i,\bar{x}_1}(c,v_1,\mathcal{W}) \wedge \cdots \wedge \exists v_{j_k} \text{ANC}_{p,i,\bar{x}_{j_k}}(c,v_{j_k},\mathcal{W}) \wedge \text{edge}^2(e,v_1,\ldots,v_{j_k})$$

Then the formula SUBGRAPH is the conjunction of S1, S2, and S3 across all i, p, and j:

$$\text{SUBGRAPH}(\mathcal{W}) : \text{S1} \wedge \bigwedge_{i,p,j} \text{S2}_{i,p,j} \wedge \text{S3}_{i,p,j}$$

Lemma 6. Let $\mathcal{G}$ be an RGG and let $G \in \mathcal{L}(\mathcal{G})$ then for each derivation tree $\mathbf{T}$ of G, $(G,\alpha_{\mathbf{T}}) \models \text{SUBGRAPH}(\mathcal{W})$.

Proof. For each $X \in N$ and for every graph $G \in \mathcal{L}_X(\mathcal{G})$, we prove that if $\mathbf{T}$ is an X-derivation tree of G then $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$.

Each production is connected and so every fully derived graph is connected, therefore each node in G is connected to some edge and S1 is satisfied.

We prove by induction on the size of $\mathbf{T}$, denoted $|\mathbf{T}|$ that S2 and S3 hold.

Base Case: If $|\mathbf{T}| = 1$, then $\mathbf{T}$ consists of a single node labelled q for some production $q : X \to G$ in P_X. There is a single anchor to deal with here, call it u. By the definition of $\mathcal{W} = \mathcal{E} \cup \mathcal{C}$, $u \in C_{0,q}$ and there is a single edge $e \in E_{0,q,j}$ for each $j \in [|\text{NT}(q)|]$ and so identifying the unique edge in each set is trivial. Each endpoint v of e must be the image of exactly one endpoint $\bar{x}$ of the edge $\bar{f}_{q,j}$ in $\textsc{rhs}(q)$ and so $\textsc{Anc}_{q,0,\bar{x}}(u, v, \mathcal{W})$ holds by Lemma 5. Therefore, S2 is satisfied. Since there is only one anchor, it also holds that for each $e \in E_{0,q,j}$ there is a unique $c \in C_{0,q}$. Therefore, S3 is satisfied and $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}_{0,q,j}(\mathcal{W})$ for each j. Since there are no other production applications to consider, this means that $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$.

Assumption:

Assume that if $|\mathbf{T}| < k$, then $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$.

Inductive Case:

Let $|\mathbf{T}| = k$. G is in $\mathcal{L}_X(\mathcal{G})$ and so the root of $\mathbf{T}$ must have label q for some $q \in P_X$ such that $q : X \to H$, H has n nonterminals $X_1, \ldots, X_n$ and $X_\eta \Rightarrow^* H_\eta$ for each $\eta \in [n]$ such that $H[X_1/H_1, \ldots, X_n/H_n] = G$. Then each H_η must have an X_η-derivation tree $\mathbf{T}_\eta$ such that $|\mathbf{T}_\eta| < k$ and so $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \textsc{Subgraph}(\mathcal{W})$ for each $\eta \in [n]$. We need to prove that $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$. Let $q_\eta \in P_{X_\eta}$ be the root of $\mathbf{T}_\eta$ for each $\eta \in [n]$.

Recall the definition of $\alpha_{\mathbf{T}}$ with respect to $\alpha_{\mathbf{T}_\eta}$ in Equations 6.1 and 6.2. For each i, p, j where $i \neq 0$ and $p \neq q_1, \ldots, q_k$, $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}_{i,p,j}(\mathcal{W})$ by the fact that $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \textsc{Subgraph}_{i,p,j}(\mathcal{W})$ for each H_η, $\eta \in [n]$. We also know that for each H_η, $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \textsc{Subgraph}_{0,q_\eta,j}(\mathcal{W})$ for some $q_\eta \in P_{X_\eta}$ and each j and so $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}_{\eta,q_\eta,j}(\mathcal{W})$ for each j. For the rest of the occurrences of q_η, $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}_{i,q_\eta,j}$ for some i and j since $(H_\eta, \alpha_{\mathbf{T}_\eta}) \models \textsc{Subgraph}_{i,q_\eta,j}$. We finally need to deal with $\textsc{Subgraph}_{0,q,j}$ for each j. The edges in G that are in $E_{0,q,j}$ for any j are precisely the terminal edges in H, and we know that each set has exactly one edge in it. Let u be in $C_{0,q}$. Then there is a single $e \in E_{0,q,j}$ for each j and each such e necessarily comes from the same production application as u did (since they are the only ones where $i = 0$) and so by Lemma 5, $\textsc{Anc}_{q,0,\bar{x}}(u, v, \mathcal{W})$ holds for each $\bar{x}$ in $\textsc{rhs}(q)$ and some v which is an endpoint of an edge in $E_{0,q,j}$ for some j. It is clear that for each anchor in $C_{i,p}$ there is a unique edge in $E_{i,p,j}$ for each j and vice versa.

Therefore, both S2 and S3 are satisfied and for each $i \in [0, |\mathrm{NT}(P)|]$, $p \in P$, and $j \in [|\mathrm{T}(P)|]$. Thi means that $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$ for each $G \in \mathcal{L}_X(\mathcal{G})$, and so $(G, \alpha_{\mathbf{T}}) \models \textsc{Subgraph}(\mathcal{W})$ for $G \in \mathcal{L}(\mathcal{G})$.

□

Proposition 5. Let $\mathcal{G}$ be an RGG and G be a graph not necessarily in $\mathcal{L}(\mathcal{G})$. Let α be a parameter assignment such that $(G, \alpha) \models \rho(\mathcal{W})$. Then if $\mathbf{T} = \tau(G, \alpha)$ is in $\mathcal{T}_{\mathcal{G}}$ then $\textsc{val}(\mathbf{T}) = G$ and so $G \in \mathcal{L}(\mathcal{G})$.

Proof. We will show that for all $X \in N$, if $(G, \alpha) \models \rho_X(\mathcal{W})$ then if $\mathbf{T} = \tau_X(G, \alpha)$ is an X-derivation tree, it is an X-derivation tree of G. This then implies that $G \in \mathcal{L}_X(\mathcal{G})$.

We prove by induction on the size of the anchor set, $|\mathcal{C}|$.

Base Case:

Let $|\mathcal{C}| = 1$ and $(G, \alpha) \models \rho_X(\mathcal{W})$. Then $(G, \alpha) \models \textsc{EdgeSets}_X(\mathcal{W})$ and $(G, \alpha) \models \textsc{Subgraph}(\mathcal{W})$. Let u be the single anchor in $\mathcal{C}$. By $\textsc{EdgeSets}_X$, there must be some $q \in P_X$ such that for each $j \in [|T(q)|]$, there is exactly one edge in G in $E_{0,q,j}$. By $\textsc{Subgraph}$, there must be some anchor in $C_{0,q}$. Since we only have one anchor, u, this means that $u \in C_{0,q}$.

There cannot be any edges in G that are in sets $E_{i,p,j}$ for $i \neq 0$ and $p \neq q$ since if there were, by $\textsc{Subgraph}$, we would also need a corresponding anchor and there are no more anchors. Therefore, $\mathbf{T} = \tau_X(G, \alpha)$ is a single node labelled q. If $\mathbf{T}$ is an X-derivation tree then $\textsc{rhs}(q)$ must be terminal, and $\textsc{Subgraph}$ established that G is isomorphic to q so $\mathbf{T}$ is an X-derivation tree of G and $G \in \mathcal{L}_X(\mathcal{G})$.

Assumption:

If $(G, \alpha) \models \rho_X(\mathcal{W}$ and $|\mathcal{C}| < k$ then if $\mathbf{T} = \tau_X(G, \alpha)$ is an X-derivation tree, it is an X-derivation tree of G.

Inductive Case:

Let $(G, \alpha) \models \rho_X(\mathcal{G})$ such that $|\mathcal{C}| = k$. Then $\mathbf{T} = \tau_X(G, \alpha)$ has k nodes. We assume that $\mathbf{T}$ is an X-derivation tree and show that it must be an X-derivation tree of G.

$(G, \alpha) \models \rho_X(\mathcal{W})$ means that $(G, \alpha) \models \textsc{EdgeSets}_x(\mathcal{W}) \wedge \textsc{Subgraph}(\mathcal{W})$. By $\textsc{EdgeSets}_X$, there must be a unique $q \in P_X$ such that for each $j \in [|T(q)|]$ there is exactly one edge e in G such that $e \in E_{0,q,j}$. By $\textsc{Subgraph}$, there is a unique $u \in C_{0,q}$ such that u anchors all of the endpoints of the edges in $E_{0,q,j}$ for each j. We fix this u for the remainder of the proof.

$\mathbf{T}$ is constructed as the output of the transducer. The nodes of $\mathbf{T}$ are defined by the anchors and their labels by sets of the form $C_{i,p}$. Therefore, $u \in C_{0,q}$ implies that there

is a node in $\mathbf{T}$ which is a 0-child (i.e. the root) and is labelled q. Let n be the number of nonterminals in $\textsc{rhs}(q)$. For $\mathbf{T}$ to be a valid X-derivation tree, the root node of $\mathbf{T}$ must have n children. Let $Y_1,\ldots,Y_n$ be the nonterminals in $\textsc{rhs}(q)$, and let $q_1,\ldots,q_n$ be the labels of the corresponding children of the root in $\mathbf{T}$. For each $i \in [n]$, we denote the subtree rooted at the i-th child of the root as $\mathbf{T}_i$. Since $\mathbf{T}$ is an X-derivation tree and the nonterminals of $\textsc{rhs}(q)$ are $Y_1,\ldots,Y_n$, $\mathbf{T}_i$ is a Y_i-derivation tree for each $i \in [n]$.

Each node in $\mathbf{T}$ is generated because of a unique anchor in G. For each $i \in [n]$, let $\mathcal{C}^i$ be the subset of $\mathcal{C}$ containing the anchors in G that become nodes in $\mathbf{T}_i$. Therefore $\mathcal{C} = \{u\} \cup \left(\bigcup_{i\in[n]} \mathcal{C}^i\right)$. For each $i \in [n]$, let H_i be the subgraph of G such that each endpoint of each edge in H_i is anchored by some anchor in $\mathcal{C}^i$. Let G' be the subgraph containing the edges whose endpoints are all anchored by u. Since $(G,\alpha) \models \textsc{Subgraph}$, the subgraphs G' and $H_1,\ldots,H_n$ partition G.

Recall the relationship between $\alpha_{\mathbf{T}}$ and $\alpha_{\mathbf{T}_\eta}$ from Equations 6.1 and 6.2. Here, we invert this relationship and define α_i for each $i \in [n]$ from α. Each α_i operates over H_i. The assignments are the same except that for the edges corresponding to the root of $\mathbf{T}_i$: if α assigned e to $E_{i,q_i,j}$, α_i assigns e to $E_{0,q_i,j}$. Similarly, for the unique anchor u_i for which $\textsc{Par}_{q,0,q_i,i}(u,u_i,\mathcal{W})$ that was assigned to C_{i,q_i} by α, is assigned to C_{0,q_i} by α_i.

We now show that for each i, $(H_i,\alpha_i) \models \rho_{Y_i}(\mathcal{W})$. By the premise of the proposition, $(G,\alpha) \models \textsc{EdgeSets}_X(\mathcal{W})$. Under the conversion from α to α_i, each edge of H_i is still assigned to exactly one set. Therefore, α_i partitions the edges of H_i. For each $e \in H_i$, if under α, e was assigned to $E_{i',p,j}$ then e must have label $\gamma_{p,j}$. Under α_i, e will be assigned to a set $E_{i'',p,j}$ where i' may not equal i''. This means that the label of e will still be $\gamma_{p,j}$. Finally, by the definition of α_i from α, $q_i \in P_{Y_i}$ is the unique production where edges in H_i are assigned to $E_{0,q_i,j}$ for each $j \in [|T(q_i)|]$. Therefore $(H_i,\alpha_i) \models \textsc{EdgeSets}_{Y_i}(\mathcal{W})$. We know that $(G,\alpha) \models \textsc{Subgraph}$. H_i is defined as being the edges whose endpoints are anchored by each of the anchors in $\mathcal{C}^i$, therefore $(H_i,\alpha_i) \models \textsc{Subgraph}$.

Therefore, $(H_i,\alpha_i) \models \rho_{Y_i}(\mathcal{W})$ for each $i \in [n]$. By the way in which H_i was defined, $\mathbf{T}_i = \tau_{Y_i}(H_i,\alpha_i)$. Since $\mathbf{T}_i$ is a Y_i-derivation tree and $\mathcal{C}^i < k$, by the inductive assumption, $\mathbf{T}_i$ is a Y_i-derivation tree of H_i.

Therefore, G is made up of a subgraph G' isomorphic to the terminal subgraph of $\textsc{rhs}(q)$ for some $q \in P_X$, and a set of edge-disjoint subgraphs $H_1,\ldots,H_n$. There are n nonterminals in $\textsc{rhs}(q)$. Let $u_1,\ldots,u_n$ be the anchors in G which correspond to the n children of the root in $\mathbf{T}$. Therefore, for each u_i for $i \in [n]$, $\textsc{Par}_{q,0,q_i,i}(u,u_i,\mathcal{W})$ holds. This means that the subgraphs $H_1,\ldots,H_n$ are connected to G' in such a way that is

permitted by the grammar and so $G = \textsc{rhs}(q)[Y_1/H_1, \ldots, Y_n/H_n]$. Therefore, $\mathbf{T}$ is an X-derivation tree of G and so $G \in \mathcal{L}_X(\mathcal{G})$.

This means that for G such that $(G, \alpha) \models \rho(\mathcal{W})$, if $\mathbf{T} = \tau(\mathcal{W})$ is in $\mathcal{T}_{\mathcal{G}}$, then $\textsc{val}(\mathbf{T}) = G$ and $G \in \mathcal{L}(\mathcal{G})$ □

Bibliography

Abend, O. and A. Rappoport
2013. Universal conceptual cognitive annotation (UCCA). In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, Pp. 228–238.

Abend, O. and A. Rappoport
2017. The state of the art in semantic representation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Pp. 77–89, Vancouver, Canada. Association for Computational Linguistics.

Allauzen, C., W. Byrne, A. de Gispert, G. Iglesias, and M. Riley
2014. Pushdown automata in statistical machine translation. *Computational Linguistics*.

Banarescu, L., C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider
2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, Pp. 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Bangalore, S. and G. Riccardi
2001. A finite-state approach to machine translation. In *Second Meeting of the North American Chapter of the Association for Computational Linguistics*.

Bar-Hillel, Y., M. A. Perles, and E. Shamir
1961. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, (14):143–172.

Bauer, D. and O. Rambow
2016. Hyperedge replacement and nonprojective dependency structures. In *Pro-*

ceedings of the 12th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+12), June 29 - July 1, 2016, Heinrich Heine University, Düsseldorf, Germany, Pp. 103–111.

Berglund, M., H. Björklund, and F. Drewes
2017. 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 2017, Umeå, Sweden, September 4-6, 2017*, Pp. 94–101.

Björklund, H., F. Drewes, and P. Ericson
2016. Between a rock and a hard place - uniform parsing for hyperedge replacement DAG grammars. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, Pp. 521–532.

Blum, J. and F. Drewes
2016. Properties of regular DAG languages. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, Pp. 427–438.

Bojańczyk, M. and M. Pilipczuk
2016. Definability equals recognizability for graphs of bounded treewidth. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Pp. 407–416.

Booth, T. and R. Thompson
1973. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, 22(5):442–450.

Bos, J.
2013. The groningen meaning bank. In *Proceedings of the Joint Symposium on Semantic Processing. Textual Inference and Structures in Corpora, JSSP 2013, Trento, Italy, November 20-22, 2013*, P. 2.

Büchi, J. R.
1960. On a decision method in restricted second-order arithmetic. *Proceedings Logic, Methodology and Philosophy of Sciences*.

Büchi, J. R. and C. Elgot
1958. Decision problems of weak second order arithmetic and finite automata, part i. *Notices of the American Mathematical Society*, P. 5;834.

Buys, J. and B. van der Merwe
2013. A tree transducer model for grammatical error correction. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task, CoNLL 2013, Sofia, Bulgaria, August 8-9, 2013*, Pp. 43–51.

Chiang, D., J. Andreas, D. Bauer, K. M. Hermann, B. K. Jones, and K. Knight
2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, Pp. 924–932.

Chiang, D., F. Drewes, D. Gildea, A. Lopez, and G. Satta
2018. Weighted dag automata for semantic graphs. *Computational Linguistics*, 44(1):119–186.

Comon, H., M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi
2007. Tree automata techniques and applications. release October, 12th 2007.

Courcelle, B.
1990. The monadic second-order logic of graphs I: recognizable sets of finite graphs. *Information and Computation*, Pp. 12–75.

Courcelle, B.
1991. The monadic second-order logic of graphs V: on closing the gap between definability and recognizability. *Theor. Comput. Sci.*, 80(2):153–202.

Courcelle, B. and J. Engelfriet
2011. *Graph Structure and Monadic Second-Order Logic, a Language Theoretic Approach*. Cambridge University Press.

D'Alembert, J.
1768. *Opuscules*, volume V.

Damonte, M., S. B. Cohen, and G. Satta
2017. An incremental parser for abstract meaning representation. In *Proceedings of*

the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers, Pp. 536–546. Association for Computational Linguistics.

Doner, J.
1970. Tree acceptors and some of their applications. *Journal of Computer and System Sciences.*, 4(5):406–451.

Drewes, F.
. On DAG languages and DAG transducers. *Bulletin of the European Association for Theoretical Computer Science*.

Drewes, F.
2017. Dag automata for meaning representations. In *Proceedings of the 15th Meeting on the Mathematics of Language (MoL 15)*, London, United Kingdom. Association for Computational Linguistics.

Drewes, F., B. Hoffmann, and M. Minas
2015. Predictive top-down parsing for hyperedge replacement grammars. In *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, Pp. 19–34.

Drewes, F., H.-J. Kreowski, and A. Habel
1997. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, ed., Pp. 95–162. World Scientific.

Drewes, F. and J. Leroux
2015. Structurally cyclic petri nets. *Logical Methods in Computer Science*, 11(4).

Droste, M. and P. Gastin
2005. *Weighted Automata and Weighted Logics*, Pp. 513–525. Berlin, Heidelberg: Springer Berlin Heidelberg.

Earley, J.
1970. An efficient context-free parsing algorithm. volume 13, Pp. 94–102, New York, NY, USA. ACM.

Engelfriet, J.
1997. Context-free graph grammars. In *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa, eds., volume 3. Springer.

Engelfriet, J. and L. Heyker
1991. The string generating power of context-free hypergraph grammars. *Journal of Computer and System Sciences*, 43(2):328–360.

Erdős, P. and A. Rényi
1959. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297.

Flickinger, D., Y. Zhang, and V. Kordoni
2012. Deepbank : a dynamically annotated treebank of the Wall Street Journal. In *Proceedings of the Eleventh International Workshop on Treebanks and Linguistic Theories (TLT11)*, Pp. 85–96, Lisbon. HU.

Gilroy, S., A. Lopez, and S. Maneth
2017a. Parsing graphs with regular graph grammars. In *Proceedings of the Sixth Joint Conference on Lexical and Computational Semantics, *SEM@ACL 2017, Vancouver, Canada, 3-4 August 2017*.

Gilroy, S., A. Lopez, S. Maneth, and P. Simonaitis
2017b. (Re)introducing regular graph languages. In *Proceedings of the 15th Meeting on the Mathematics of Language (MoL 15)*, London, United Kingdom. Association for Computational Linguistics.

Groschwitz, J., M. Lindemann, M. Fowlie, M. Johnson, and A. Koller
2018. Amr dependency parsing with a typed semantic algebra. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Pp. 1831–1841. Association for Computational Linguistics.

Hajič, J., E. Hajičová, J. Panevov, P. Sgall, O. Bojar, S. Cinková, E. Fučíková, M. Mikulová, P. Pajas, J. Popelka, J. Semecký, J. Šindlerová, J. Štěpánek, J. Toman, Z. Urešová, and Z. Žabokrtský
2012. Announcing prague czech-english dependency treebank 2.0. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey. European Language Resources Association (ELRA).

Hopcroft and Ullman
1979. *Introduction to automata theory, languages and computation*. Addison-Wesley.

Jones, B. K., J. Andreas, D. Bauer, K. M. Hermann, and K. Knight
2012. Semantics-based machine translation with hyperedge replacement grammars. In *COLING 2012, 24th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, 8-15 December 2012, Mumbai, India*, Pp. 1359–1376.

Kamimura, T. and G. Slutzki
1981. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51.

Koller, A. and M. Kuhlmann
2011. A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*, Pp. 2–13, Dublin, Ireland. Association for Computational Linguistics.

Kuhlmann, M. and S. Oepen
2016. Towards a catalogue of linguistic graph banks. *Computational Linguistics*, 42(4):819–827.

Lautemann, C.
1990. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421.

Li, Y., O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia
2018. Learning deep generative models of graphs. *CoRR*, abs/1803.03324.

Lyu, C. and I. Titov
2018. Amr parsing as graph prediction with latent alignment. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Pp. 397–407. Association for Computational Linguistics.

Matheja, C., C. Jansen, and T. Noll
2015. *Tree-Like Grammars and Separation Logic*, Pp. 90–108. Cham: Springer International Publishing.

McAllester, D.

2002. On the complexity analysis of static analyses. *Journal of the Association for Computing Machinery*, 49(4):512–537.

McNaughton, R. and S. A. Papert

1971. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press.

Mohri, M., F. C. N. Pereira, and M. Riley

2008. Speech recognition with weighted finite-state transducers. In *Handbook on Speech Processing and Speech Communication, Part E: Speech recognition*, L. Rabiner and F. Juang, eds. Springer.

Peng, X., L. Song, and D. Gildea

2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the 19th Conference on Computational Natural Language Learning, CoNLL 2015, Beijing, China, July 30-31, 2015*, Pp. 32–41.

Post, E. L.

1946. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268.

Quernheim, D. and K. Knight

2012. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*, SSST-6 ’12, Pp. 76–85, Stroudsburg, PA, USA. Association for Computational Linguistics.

Reiter, F.

2014. Distributed graph automata. Master’s thesis.

Reutenauer, C.

1990. *The Mathematics of Petri Nets*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Roark, B. and R. Sproat

2007. *Computational Approach to Morphology and Syntax*. Oxford University Press.

Shieber, S. M., Y. Schabes, and F. C. N. Pereira
1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1&2):3–36.

Song, L., X. Peng, Y. Zhang, Z. Wang, and D. Gildea
2017. Amr-to-text generation with synchronous node replacement grammar. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Pp. 7–13. Association for Computational Linguistics.

Thomas, W.
1991. *Automata, Languages and Programming: 18th International Colloquium Madrid, Spain, July 8–12, 1991 Proceedings*, chapter On logics, tilings, and automata, Pp. 441–454. Berlin, Heidelberg: Springer Berlin Heidelberg.

Trakhtenbrot, B.
1966. Finite automata and logic of one-place predicates. *American Mathematical Society Translation*, 59:23–55.

van Noord, R. and J. Bos
2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *CoRR*, abs/1705.09980.

Vasiljeva, I., S. Gilroy, and A. Lopez
2018. The problem with probabilistic dag automata for semantic graphs.

Vijay-Shanker, K., D. J. Weir, and A. K. Joshi
1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th Annual Meeting on Association for Computational Linguistics*, ACL '87, Pp. 104–111, Stroudsburg, PA, USA. Association for Computational Linguistics.

Wang, C., N. Xue, and S. Pradhan
2015. A transition-based algorithm for amr parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Pp. 366–375. Association for Computational Linguistics.

Woods, W. A.
1979. *Semantics for a question-answering system.* New York : Garland Pub. Originally presented as the author's thesis, Harvard, 1967.