Methods for taking semantic graphs apart and putting them back together again

Jonas Groschwitz

PhD thesis developed at
the Philosophische Fakultät
at Saarland university
and
the Department of Computing
at Macquarie University

March 20, 2019

Contents

List of Figures

List of Tables

Summary

This thesis develops the AM dependency parser, a semantic parser for Abstract Meaning Representation (AMR, Banarescu et al. (2013)) that owes its strong performance to its effective combination of neural and compositional methods. Neural networks have proven to be enormously effective machine learning tools for natural language processing. Compositionality as a linguistic principle has a strong tradition in semantic construction. However, both approaches have distinct challenges. Pure neural models are data hungry, since they have no prior knowledge of the inherent structure in language. Compositional approaches have robustness issues and suffer from the ambiguity of latent structural information in the training data.

This thesis combines the strengths of both worlds to address these challenges. The AM dependency parser drops the restrictive syntactic constraints of classic compositional approaches, instead relying only on semantic types and meaningful semantic operations as structural guides. The ability of neural networks to encode contextual information allows the parser to make correct decisions in the absence of hard syntactic constraints.

Consequently, the thesis focuses on *terms* for semantic representations, which are algebraic 'building instructions'. The thesis first examines the suitability of the HR algebra (a general tool for building graphs, Courcelle and Engelfriet (2012)) for this purpose. It then develops the linguistically motivated AM algebra, that proves much better suited for the purpose. Representing the terms over the AM algebra as dependency trees further simplifies the semantic construction. In particular, the move from the HR algebra to the AM algebra and then to AM dependency trees drastically removes the ambiguity of latent structural information required for training the model.

In conclusion, the AM dependency trees yield a simple semantic parser, where neural tagging and dependency models predict interpretable, meaningful operations that construct the AMR.

Statement of Originality

This thesis is being submitted to Macquarie University and Saarland University in accordance with
the Cotutelle agreement dated 01-08-2017.

To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

(Signed) [illegible] Date: 30-09-2018

Jonas Groschwiz

Thanks Meaghan,
for opening the world of linguistics to me.

Acknowledgements and Thanks

First and foremost, I'd like to thank my supervisors Alexander Koller and Mark Johnson. Thank you for all your support during the last few years. You were both kind and demanding, all at the right times. You always treated me as a human first, especially during the more difficult parts of this process when I really needed it. Your guidance, both professionally and personally, has been invaluable to me and will stay with me for a long time.

On a more practical note, I'd like to very much thank my co-authors for the papers this thesis is based on, Meaghan Fowlie, Mark Johnson, Alexander Koller, Matthias Lindemann and Christoph Teichmann. Thank you for the very enjoyable collaborations, and for your permission to use our papers in this thesis.

Further thanks go to Meaghan Fowlie, David M. Howcroft, Antoine Venant and Tobias Sommer for proofreading my thesis. Your feedback was enormously helpful. Also I'd like to thank Prof. Dr. Manfred Pinkal for an illuminating discussion on compositionality. And a particular shout-out to Martín Villalba for letting me use his neat LaTeXthesis template. All mistakes in this thesis remain my own, of course.

A bit more personal: thank you to ChilledCow on YouTube for keeping me working; and ,more importantly, thank you to the cast of Critical Role for continuously reminding me what being human is about.

And finally, a heartfelt thank you to Meaghan, Elvira, Wigbert, Rebecca, Sven and Melina for their patience and support while I was preparing the thesis. It meant a lot.

I'll leave it at that, even though I also owe thanks to many more acquaintances, friends, teachers, talk-givers, the list goes on. I appreciate all of it and hope I can give some of it back to whoever needs it in the future.

The work in this thesis has been supported by DFG grant KO 2916/2-1 and a Macquarie University Research Excellence Scholarship.

1

Introduction

The topic of this thesis is semantic parsing, more specifically AMR parsing. The thesis combines ideas from traditional semantic construction with current neural methods, developing a new, simple model that expresses operations over a semantic algebra as dependencies on the sentence. The model yields state of the art parsing performance in practice.

1.1 Semantic parsing

Semantic parsing is the task of translating a sentence into a meaning representation. Such a meaning representation can take different forms, depending on what one wants to do with it. The commonality is that the meaning representation captures (at least some aspects of) the sentence's meaning in a more abstract and structured form than present in the sentence itself. Such a representation can then be further processed in a downstream task, such as answering questions, extracting information from a text, or analyzing the sentiment of a twitter message. This makes semantic parsing an important part of the natural language processing toolkit.

At the same time, semantic parsing is a very challenging task. There is an infinite number of possible sentences, varying greatly in content and structure, with a large number of ambiguities, idiosyncrasies and inconsistencies. A common approach to address this variance is to not create a full parser by hand, but instead use corpora of sentences annotated with meaning representations to have the parser learn its parameters from the corpus; this thesis follows in this tradition.

The semantic representation formalism that this thesis works with is Abstract Meaning Representation (AMR, Banarescu et al. (2013)). AMR is a general purpose meaning representation for the English language. It represents meanings as graphs, such as the one in Figure 1.1 for the sentence *The raven wants to learn.* At its core, AMR represents the "who did what to whom" of a sentence, with nodes representing concepts and edges the relations between them. In this example, the two ARG0 edges indicate that the raven is both the wanter and the learner, and the ARG1 edge indicates that the learning is what the raven wants. There are some semantic phenomena that AMR does not encode, such as many aspects of quantification. This makes it easier to annotate corpora with AMR and allows semantic parsing research to focus on a more concentrated set of issues.

Due to the availability of large annotated corpora, AMR has proven a fertile research ground for semantic parsing. Applications of AMR include e.g. information extraction (Wang et al. (2017)), paraphrase detection (Issa et al. (2018)) and summarization (Dohare et al. (2018)). But maybe no less importantly, the general purpose nature of AMR, combined with the fact that AMR covers many common yet challenging phenomena (such as long distance semantic dependencies, or normalizing paraphrases to the same meaning), means that solutions for many problems in AMR parsing may well be applicable to parsing into other semantic formalisms as well. Thus, advancing the state of AMR parsing is likely to advance the state of semantic parsing as a whole.

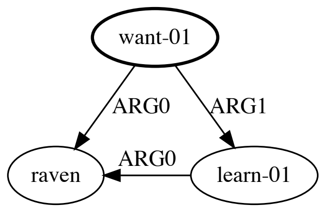
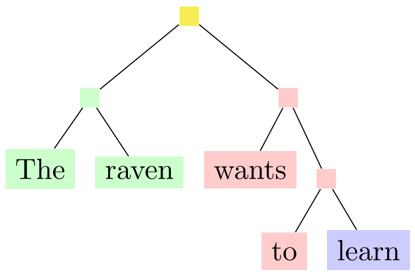


Figure 1.1: AMR for the sentence *The raven wants to learn.*

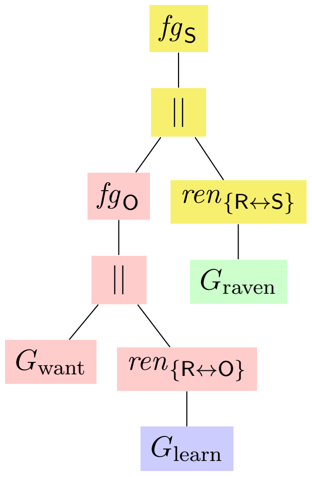
1.2 Linguistic structure in AMR parsing

Since semantic parsing is all about understanding natural language, one might assume that most semantic parsers draw heavily on the linguistic theory established over the course of the past century. However, matching a current trend in computational linguistics, the most successful AMR parsers of recent years (e.g. Foland and Martin (2017), van Noord and Bos (2017)) use essentially no linguistic theory, instead relying on powerful machine learning techniques – specifically, neural networks. The model of van Noord and Bos (2017) simply predicts a string representing the AMR character by character, and Foland and Martin (2017) predict graph nodes first and then draws edges between them, without making explicit use of e.g. the sentence's syntactic structure.

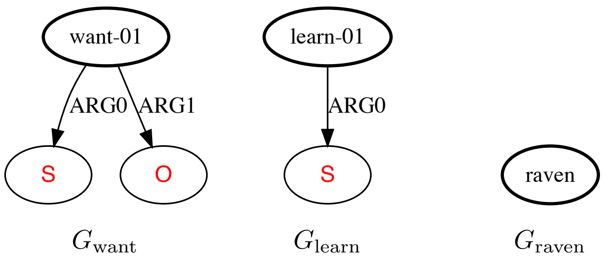
And yet, adding linguistic knowledge to guide the machine learning modules, to give them a 'head start' in their training process, remains an attractive idea. After all, AMRs are not just nodes and edges, or sequences of characters. Rather, they relate to the sentence and its syntactic structure in often direct ways.



(a) (Simplified) syntax structure.



(b) Term over the HR algebra that evaluates to the AMR in Figure 1.1.



(c) The graph fragments used in (b).

Figure 1.2: Syntax structure and semantic term for *The raven wants to learn*, with colors indicating matching parts.

The concept of *compositionality* captures this relation well. The fundamental assumption of compositionality is that there is a structurally consistent map between the *syntactic surface structure* of a sentence and the corresponding *semantic term*. An example of the syntactic structure is the (simplified) constituent tree in Figure 1.2(a). A semantic term is a tree of operations such as the one in Figure 1.2(b). Such a term is a 'building instruction' for the meaning representation, where operations combine smaller graph fragments, such as the ones in Figure 1.2(c), into the final AMR. As we will see later, the term in Figure 1.2(b) evaluates to the AMR in Figure 1.1. We will formally define terms in Chapter 2. The operations in this term here, i.e. $||$, $ren_{\mathsf{R}\leftrightarrow\mathsf{O}}$ etc., are those of the *HR algebra* (Courcelle and Engelfriet (2012)) which is a standard tool that plays a central role in this thesis.[1] The red markers S and O in the graph fragments in Figure 1.2(c) are part of the HR algebra, and tell it where and how the graph fragments can combine. We give a detailed introduction to the HR algebra in Chapter 2.

An example of a structurally consistent map between the syntax tree and the semantic term, as assumed by compositionality, is indicated by the colors in Figure 1.2. Then, if one knows the syntactic structure of a sentence, and how each part of it maps to a piece of the semantic term, one can from that construct the full semantic term and thus the meaning representation.

One classic compositional approach to semantic parsing is that of synchronous grammars. A *grammar* can be characterized as a set of rules that describe the syntax of a sentence, and a *synchronous* grammar for semantic parsing associates each rule with a partial semantic term, building the full semantic term synchronously with the syntax when parsing the sentence. There are many variants of synchronous grammars, but they share common principles. Two example grammar rules could look something like this:

NP → *the raven*	G_{raven}
S → NP VP	$fg_{\mathsf{S}}\left(x_2 \mid\mid ren_{\{\mathsf{R}\leftrightarrow\mathsf{S}\}}\left(x_1\right)\right)$

The two rows each describe a rule and correspond to the green and yellow segments in Figure 1.2 respectively. In the first row, the left column indicates that this rule produces the string *the raven*, with *syntactic category* NP, i.e. a noun phrase. The right column introduces the *raven* node G_{raven} of Figure 1.2(c) into the semantic term.

In the second row, the left column indicates that the rule takes two arguments and concatenates them, with one argument having category NP such as *the raven* as

[1]The name is due to the close relation to Hyperedge Replacement Grammar (HRG, Drewes et al. (1997)), as Courcelle and Engelfriet (2012) discuss.

we just saw, and the other having the category VP. This second argument could be the phrase *want to learn*, i.e. the result of the red and blue segments in Figure 1.2. The right column corresponds to the yellow sequence of semantic operations in Figure 1.2(b). In conclusion, synchronous grammars have rules that represent pairs of syntactic and semantic operations, and how the rules can be combined is typically controlled with syntactic categories.

AMR parsers based on synchronous grammars already exist, for example, Artzi et al. (2015) is based on Combinatory Categorial Grammar (CCG, Steedman (2000)), and Peng et al. (2015) is based on Hyperedge Replacement Grammar (HRG, Drewes et al. (1997)). While the parser of Artzi et al. (2015) outperformed other early AMR parsers and Peng et al. (2015) also showed solid performance, these synchronous grammar approaches did not produce much follow-up work and the state of the art has been pushed quite significantly since then. As a result, there is no current competitive AMR parser based on synchronous grammars.

To summarize, we have two kinds of existing approaches, one with minimal to no linguistic structure, instead heavily relying on machine learning and neural networks. The other kind are the heavily structured synchronous grammars. Both approaches face distinct challenges. A useful perspective to understand those challenges is the following. Several principles of how meanings of words combine have been observed in language. However, there is much variance in the *instantiations* of these principles, i.e. in how exactly e.g. word order and lexical information affect semantic construction in different contexts. Encoding these instantiations manually is very labor intensive. Thus, the idea of adding linguistic structure to a semantic parser is the following: to encode the general principles in the framework of our learning algorithm, and learn the different instantiations and variations from data.

This observation that there are universal linguistic principles, distinct from their specific instantiations within and across languages, has a long history in linguistic theory – see for example the notion of Principles and Parameters in Universal Grammar (e.g. Chomsky (1981)). The principle of compositionality is one of these principles, and examining different compositional approaches reveals common mechanisms that repeat throughout the English language. We will discuss these principles in more detail in Chapter 3.

Under this perspective of encoding linguistic principles directly but learning their instantiations, two challenges of the neural, less structured models become clear.

Challenge 1: Hunger for data. The neural approaches don't encode these general linguistic principles. Thus, they have to learn the principles too, often making the models particularly data hungry. For example, the purely neural approach of van Noord and Bos (2017) requires the use of extra training data, generated by other AMR parsers, to achieve good performance.

Challenge 2: Limited linguistic extensibility. Neural networks are a quite opaque formalism – they can be characterized as sequences of operations on high dimensional vector spaces. Thus, it is unclear how to add these linguistic principles into, say, a sequence-to-sequence neural approach like the one of van Noord and Bos (2017).

Synchronous grammar based parsers do follow this idea of encoding principles directly, and learning the instantiations from data. They do this by specifying the general form of the grammar rules, but induce the specific rules themselves from the data. Still, these parsers face two challenges of their own.

Challenge 3: Robustness. The mechanisms of synchronous grammars for building AMRs are more complex and restrained when compared to the more machine learning based models. Where van Noord and Bos (2017) simply predicts a linear representation of an AMR character by character, or Foland and Martin (2017) predicts graph nodes first and then just draws edges between them, the grammar based models use large sets of rules featuring many hard constraints on how the rules can combine. In other words, the instantiations of the general principles that synchronous grammars learn only apply to very specific situations. This makes the systems susceptible to irregularities in the data, such as grammar mistakes in a sentence, or rare syntactic phenomena and rare words. While such irregularities often only cause one wrong node or edge in the less structured models, for grammar based models they can completely throw off the delicate derivation process.

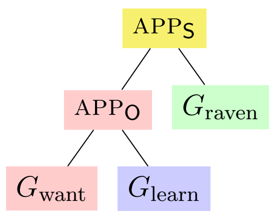
Challenge 4: Structural ambiguity in the training data. To train a synchronous grammar on a corpus, one needs to know the syntax trees and semantic terms for all training examples – after all, these are the structures a synchronous grammar must learn to predict. However, the AMR corpora are not annotated with these structures – the syntactic and semantic terms are *latent*. To train a grammar then, one must come up with such structures for the whole corpus. The problem is, as e.g. Peng et al. (2015) note, there are gigantic numbers of terms that could produce any given AMR. Manually annotating the corpus with terms and alignments would take enormous labor resources. Artzi et al. (2015) and Peng et al. (2015) therefore use heuristic and statistical methods to select the terms they use for training, but the problem is far from solved.

1.3 What this thesis is about

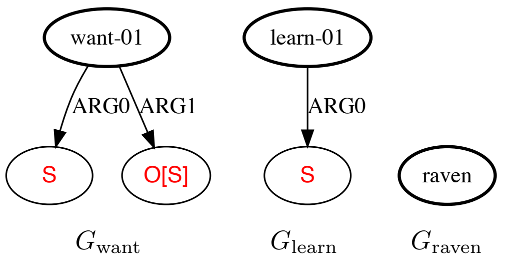
This thesis develops a new semantic parsing model for AMR that combines the strengths of the existing neural and compositional approaches and addresses the challenges listed above. A key insight is that many of the principles that compositional approaches encode lie not in the relation between the syntax and the semantic terms, but in the semantic terms alone. We will see this in detail in Chapter 3 on background in semantic parsing. As a consequence, we still face the training data issue, i.e. Challenge 4 above. We address this issue first, and develop the new parser later based on the insights we gained along the way.

HR decomposition automata. The above mentioned HR algebra is a general tool for constructing graphs, and Koller (2015) suggests it as particularly suited for semantic construction. The first contribution of this thesis is a method to compute *decomposition automata* for the HR algebra, which are compact representations of all terms over the HR algebra that evaluate to a given AMR. Such a compact representation of the set of terms will be crucial for training any parser based on the HR algebra: it allows generating terms for the AMR in the corpus, e.g. via statistical methods as used by Peng et al. (2015). Examining these compact representations, the decomposition automata, will also help us understand how much ambiguity there is when expressing an AMR with an HR term, i.e. how much of a problem the above mentioned Challenge 4 is for the HR algebra. Computing the decomposition automata turns out to be a complex task. Thus the technical challenge that we address here is to compute the decomposition automata efficiently. We find that in fact, there are so many different HR terms for each AMR that finding consistent terms for training is infeasible with purely statistical methods. That is, Challenge 4 applies here as well. This is what we address next.

AM algebra. Examining the HR terms for AMR more closely, we find that since the HR algebra is designed to create general graphs, it has many ways of building an AMR that are undesirable when we consider the AMR as a semantic graph with meaningful structure. In other words, there is much unnecessary ambiguity when choosing an HR term for a given AMR. As a solution, this thesis presents the new Apply-Modify (AM) algebra. Its operations wrap linguistically motivated sequences of HR operations into single operations, and adds a simple type system that controls which operations are allowed when.



(a) AM term.



(b) The graph fragments used in (a).

Figure 1.3: Term over the AM algebra that evaluates to the AMR for *The raven wants to learn* in Figure 1.1. The colors indicate parts matching with the terms in Figure 1.2.

A term over the AM algebra that evaluates to the AMR in Figure 1.1 is shown in Figure 1.3(a). It uses the *application* operation APP of the AM algebra. The idea is that the graph fragments have 'slots', marked with the red letters in Figure 1.3(b), and that the application operation fills these slots.

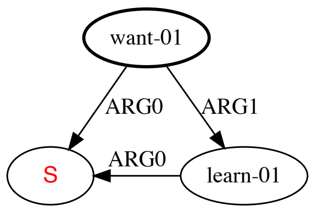


Figure 1.4: Partial result of the term in Figure 1.3(a), after APP$_\mathsf{O}$ $(G_\text{want}, G_\text{learn})$.

For example, The APP$_\mathsf{O}$ operation in APP$_\mathsf{O}$ $(G_\text{want}, G_\text{learn})$ plugs the graph fragment G_learn into the O slot of G_want (marked with the red $\mathsf{O[S]}$ in G_want in Figure 1.3(b)). The result is shown in Figure 1.4. Both G_want and G_learn have an S slot here, and these S slots are *unified* into one during application, as shown in Figure 1.4. Thus, the result of APP$_\mathsf{O}$ $(G_\text{want}, G_\text{learn})$ has a single S slot, and the APP$_\mathsf{S}$ operation in the term in Figure 1.3(a) fills this slot with the graph G_raven, to obtain the complete AMR in Figure 1.1.

This kind of argument application, filling and unifying 'slots', is a classic method in compositional semantic construction, used for example in Lexical Function Grammar (LFG; Kaplan et al. (1982)) and the Minimal Recursion Semantics algebra (MRS; Copestake et al. (2001)) for Head-driven Phrase Structure Grammar (HPSG; Pollard and Sag (1994)).

We control when unification occurs with the type system. The "[S]" marker at the O slot of G_want (written as $\mathsf{O[S]}$ in Figure 1.3(b)) ensures that the argument that fills the O slot, here G_learn, has an open S slot itself, guaranteeing that the unification occurs.

The type system and the higher-level operations work together to effectively restrict the set of possible terms for a given AMR. We need to strike a careful balance here. If we restrict the algebra too much, then we get coverage problems, i.e. there are too many AMRs for which we cannot find a good term at all. But if we don't restrict the algebra enough, we end up with the same complexity problems we observed in the HR algebra. Qualitative and quantitative evaluations in this thesis support the claim that the AM algebra successfully strikes that balance.

AM dependency trees. With this work done on the training data issue (Challenge 4) and the AM algebra at our disposal, we now introduce the new parsing model. It predicts AM operations directly, in the form of a dependency tree like the one in Figure 1.5. This dependency tree assigns graph fragments to some of the words (here G_{raven}, G_{want} and G_{learn}, written below the sentence), and specifies which operations occur between them (the arrows above the sentence). Such a dependency tree is an *underspecified* representation of an AM term, because it specifies which operations occur, but not the order. For example, the dependency tree here does not specify whether the APP_{S} or the APP_{O} operation is executed first. We show that although the *AM term* is underspecified, the AMR is not: that all well-typed terms corresponding to one dependency tree evaluate to the same AMR. Thus, knowing the dependency tree is enough to uniquely specify an AMR as evaluation result.

We now turn to the parsing pipeline. To predict the dependency trees, we use standard neural techniques for supertagging (to predict the graph fragments, see e.g. Lewis and Steedman (2014)) and for dependency parsing (see e.g. Kiperwasser and Goldberg (2016)). The full pipeline has the following steps.

Training

1. Generate AM dependency trees for the AMRs in the training data. In fact, there are now even fewer dependency trees describing an AMR than there are AM terms, so few in fact that we can just pick an arbitrary dependency tree and get consistent training data.
2. Train the neural supertagger and dependency models to predict the graph fragments and edges of the dependency trees in the training data.

Prediction

1. Predict scored lists of potential graph fragments for each word, and scores for all possible edges (i.e. between each pair of words) with the neural models, see Figure 1.6 on the left.
2. Use a typed decoder to find the best well-typed dependency tree, according to the scores of the neural models, see Figure 1.6 on the right.

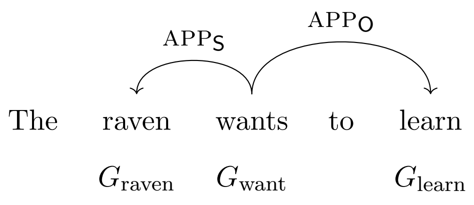


Figure 1.5: AM dependency tree for the running example *The raven wants to learn.*

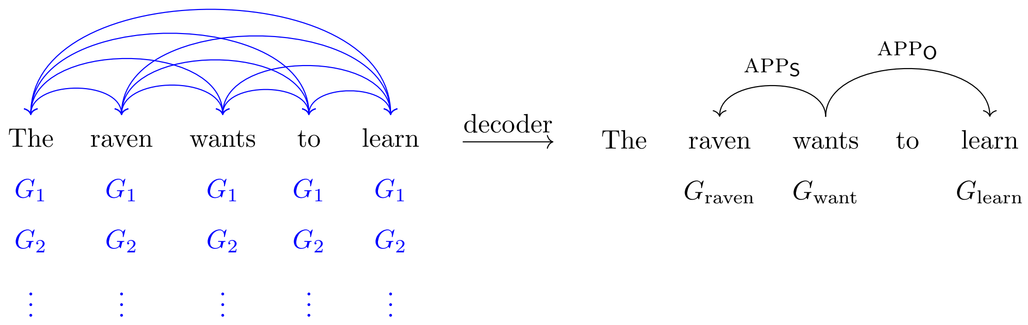


Figure 1.6: The neural model predicts scores for all possible edges and supertags, i.e. for all blue edges and supertags on the left. The decoder then finds the best AM dependency tree according to the scores.

3. Evaluate the dependency tree to get an AMR.

Since the dependency model predicts the semantic operations directly, it is simpler, and more flexible and robust than a synchronous grammar. At the same time, the linguistically motivated AM operations provide structure to the parser. We thus still guide the parser with linguistic principles, just more gently.

The result is a parser that reaches 71.0 Smatch score on the LDC2017T10 dataset, a state of the art result.

1.4 Plan of thesis and contributions

Chapter 2. Background: Semantic graphs. This thesis brings together a variety of mathematical and formal notions as background. Chapter 2 describes AMR in more detail, introduces the HR algebra of Courcelle and Engelfriet (2012) whose operations we use to construct AMRs (c.f. the term in Figure 1.2(b)), and introduces decomposition automata (or, more generally, tree automata), a compact representation of terms that will be useful throughout the thesis. Chapter 2 also introduces the technical notations we use.

Chapter 3. Background: Semantic parsing. Where Chapter 2 introduced technical background, Chapter 3 gives an overview on previous work on semantic parsing. We get to know methods from compositional semantic parsing, illustrated by the IRTG formalism combined with the HR algebra. These methods will serve as inspiration throughout the thesis, and are very much the foundation for the following chapters. Chapter 3 also discusses related work in AMR parsing in more detail, and situates the work of this thesis within that scope.

Chapter 4. S-Graph Decomposition. This chapter describes how to compute the decomposition automata for the HR algebra efficiently, and provides asymptotic and empirical evaluation. The content of this chapter is based on Groschwitz et al. (ACL 2015).

Key contributions:

- Fast bottom-up and top-down algorithms for computing the HR decomposition automata.
- Empirical runtimes on the related graph parsing task improved the state of the art by orders of magnitude, and were further improved in Groschwitz et al. (ACL 2016).

Chapter 5. The AM Algebra. This chapter introduces the AM algebra. The AM algebra was first published in Groschwitz et al. (IWCS 2017). This chapter is based on that publication, with several new ideas added.

Key contributions:

- The AM algebra, a linguistically motivated algebra for semantic graphs.
- A qualitative analysis of the AM algebra's suitability for several language phenomena.
- A method for computing the decomposition automata for the AM algebra, including an evaluation on an AMR corpus.

Chapter 6. AM dependency parsing. This chapter introduces the AM dependency trees, as well as the parser for this dependency model. The chapter is based on Groschwitz et al. (ACL 2018).

Key contributions:

- AM dependency trees as a simple, sound and flexible model for semantic construction.
- The AM dependency parser, that combines the structure of the AM algebra with neural machine learning techniques to yield state of the art performance in AMR parsing.

1.5 Scope of the thesis

The goal of this thesis is to develop a competitive neural compositional AMR parser that addresses the four challenges set out above. In particular, the thesis aims to get the basics right. While throughout the thesis we will encounter related problems that are complex research topics in their own right, solving these problems is beyond the scope of this work. Such out-of scope topics are in particular coreference, ellipsis and the Zipfian tail of more specific compositional phenomena.

2

Background: Semantic graphs, and building them piece by piece

In this chapter, we examine AMR in detail and introduce the HR algebra of Courcelle and Engelfriet (2012) (henceforth C&E), whose terms we use to describe AMRs. We also take a look at tree automata, that allow us to represent large sets of terms compactly. The goal of this chapter is thus to familiarize the reader with the semantic representation we are concerned with, and to establish foundational methods for building these graphs from smaller pieces.

Section 2.1 introduces some basic notation. Section 2.2 formally defines graphs, and Section 2.3 introduces AMR in more detail. We introduce the HR algebra, as well as background on algebras and terms in general, in Section 2.4. The chapter concludes with an introduction to tree automata.

2.1 Basics

Before we look at semantic graphs, let us get some basic notation down.

- For a set M, we denote its **power set** (the set of all subsets of M) as $\mathcal{P}(M)$.
- We write the **identity function** as id.
- For a function $f : A \to B$, we write its **domain** as $D(f) = A$, and its **image** as image $I(f) = B$.

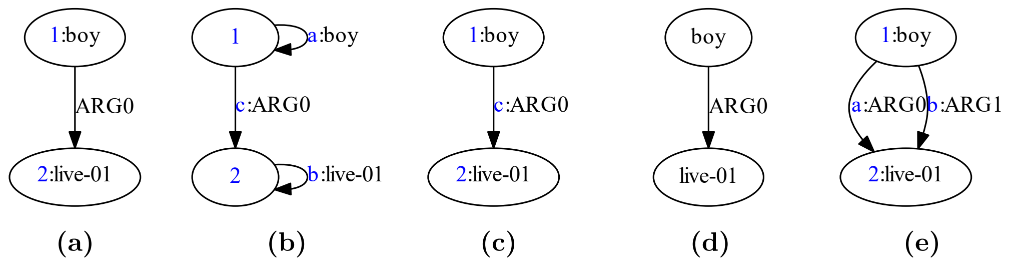


Figure 2.1: Example graphs.

- We **restrict** a function f to a set $A \subseteq D(f)$ by writing $f|_A$; this is the function identical to f, but with domain A.

- Given two functions f and g such that the image of g is in the domain of f, i.e. $I(g) \subseteq D(f)$, we define their **concatenation** $f \circ g$ element wise as $(f \circ g)(x) = f(g(x))$.

- Given two functions f and g, we write their **union** as $f \cup g$. This is only defined if f and g agree where their domains overlap, i.e. if for all x in $D(f) \cap D(g)$, we have $f(x) = g(x)$. Then, the union $f \cup g$ is a function on the union of domains, $D(f) \cup D(g)$, with

$$(f \cup g)(x) = \begin{cases} f(x) & \text{if } x \in D(f), \\ g(x) & \text{otherwise.} \end{cases}$$

- Given an alphabet of symbols Σ we denote the set of all **sequences** (or **strings**) over Σ with Σ^*, and the empty sequence with ϵ.

2.2 Graphs

In this thesis, we work with graphs a lot. There are multiple ways of defining a graph, and it will be necessary to strike a balance between flexibility and simplicity here.

The most straightforward way of defining a labeled graph is the following:

Definition 2.1. A *simply labeled directed simple graph* for a set of node labels K and edge labels Λ is a tuple $G = (V_G, E_G, \kappa_G, \lambda_G)$, where V_G is a set of *vertices*, $E_G \subset V_G \times V_G$ is the set of edges, and $\kappa_G : V_G \to K$ and $\lambda_G : E_G \to \Lambda$ are (partial) node- and edge labeling functions.

Such a graph is shown in Figure 2.1(a). In examples, such as Figure 2.1, we generally denote nodes with blue numbers (we omit the colors when we refer to the nodes in text). The black texts are the labels.

However, this formalization of graphs is suboptimal for this thesis. On the one hand, AMRs do not use loops, so we can encode node labels as loops instead; see Figure 2.1(b) for an example. This will simplify the reasoning in formal parts of this thesis, since we then can reason about edges only. Further, we will need to consider edges in multiplicity, i.e. *multigraphs*. This is for two reasons. One, AMRs (which we introduce in more detail in Section 2.3) can be multigraphs, for example Figure 2.1(e) shows an AMR for *The wizard likes himself*, where *wizard* is both the ARG0 and ARG1 of *like-01*, i.e. both the liker and the object of affection. The second reason is that the graph merging operations of the HR algebra, which we will use throughout the thesis, use multigraphs (see Section 2.4). We thus don't define edges just through their source and target node, but give them their own identities (blue lowercase letters in examples, such as in Figure 2.1(b)). We use the following definition of graphs throughout the thesis.

Definition 2.2. An *edge-labeled directed multigraph* for a set of edge labels Λ is a quadruple $G = (V_G, E_G, vert_G, \lambda_G)$ where V_G is a set of *vertices*, and E_G is a set of *edges*. The function $vert_G : E_G \to V_G \times V_G$ assigns a pair of vertices (u, v) to each edge, the edge is then considered to go from u to v. The edge labeling function $\lambda_G : E_G \to \Lambda$ assigns to each edge a label in Λ.

Usually though, while we formally encode node labels as loops, in examples we present them directly as node labels to improve readability. For example, we would use the notation of Figure 2.1(c) rather than the one in Figure 2.1(b). We also usually refer to them as node labels in text, to keep the language simple.

Since much of this thesis relies on the HR algebra of Courcelle and Engelfriet (2012) (C&E), it is worth mentioning the graph definition from there (Definitions 2.9 and 2.11 of C&E, with minor changes):

Definition 2.3. A *multi-labeled directed multigraph* for a set of node labels K and edge labels Λ is a quintuple $G = (V_G, E_G, vert_G, \kappa_G, \lambda_G)$ where V_G, E_G,$vert_G : E_G \to V_G \times V_G$ and $\lambda_G : E_G \to \Lambda$ are as in Definition 2.2. The node labeling function $\kappa_G : V \to \mathcal{P}(K)$ assigns to each node a (possibly empty) set of labels from K.

Remark 2.4. It is straightforward to translate between our definitions in some cases. Take Definitions 2.1 and 2.3. For any simply labeled directed simple graph G, let its *canonical multigraph* be the tuple $G' = (V_{G'}, E_{G'}, vert_{G'}, \kappa_{G'}, \lambda_{G'})$ with

$$\begin{aligned} V_{G'} &= V_G \\ E_{G'} &= E_G \\ vert_{G'} &= \mathsf{id} \\ \lambda_{G'} &= \lambda_G \end{aligned}$$

and with $\kappa_{G'}(v) = \{\kappa_G(v)\}$ if $\kappa_G(v)$ is defined, and $\kappa_{G'}(v) = \emptyset$ otherwise. Conversely, let G be a multi-labeled directed multigraph without multiple edges (i.e. $vert_G$ is injective) and where each node has at most one label. Then its *canonical simple graph* is the tuple $G' = (V_{G'}, E_{G'}, \kappa_{G'}, \lambda_{G'})$ with

$$\begin{aligned} V_{G'} &= V_G \\ E_{G'} &= vert_G\,(E_G) \\ \lambda_{G'} &= \lambda_G \circ vert_G^{-1} \end{aligned}$$

and with $\kappa_{G'}(v) = a$ if $\kappa_G(v) = \{a\}$ for some $a \in K$, and $\kappa_{G'}(v)$ undefined if $\kappa_G(v) = \emptyset$ (no other cases are possible).

Similarly, we can translate between Definitions 2.3 and 2.2 in some cases. Expressing an edge-labeled directed multigraph as a multi-labeled graph is always possible: for any edge-labeled directed multigraph G, let its *canonical multi-labeled graph* be the tuple $G' = (V_{G'}, E_{G'}, vert_{G'}, \kappa_{G'}, \lambda_{G'})$ with

$$\begin{aligned} V_{G'} &= V_G \\ E_{G'} &= \{e \in E_G \mid u \neq v \text{ for } (u, v) = vert_G(e)\} \\ vert_{G'} &= vert_G|_{E_{G'}} \\ \lambda_{G'} &= \lambda_G|_{E_{G'}} \end{aligned}$$

and with $\kappa_{G'}(v) = \{\lambda_G(e) \mid vert_G(e) = (v, v)\}$ for all $v \in V_{G'}$. In other words, we simply interpret all loops as node labels instead.

However, inverting this procedure to express a multi-labeled graph G as an edge-labeled directed multigraph is only possible if G has no loops. Let G be a multi-labeled directed multigraph with node labels in K and without loops (i.e. for all

$(u, v) \in I\left(vert_G\right)$, $u \neq v$), and let

$$f : V_G \times K$$

be injective to a set A that is disjoint to E_G (f will create new edges for us). Then a *canonical edge-labeled graph* for G is the tuple $G' = (V_{G'}, E_{G'}, vert_{G'}, \lambda_{G'})$ with

$$\begin{aligned} V_{G'} &= V_G \\ E_{G'} &= E_G \cup \{f\left(v, \ell\right) \mid v \in V_G, \ell \in \kappa_G(v)\} \end{aligned}$$

and with $\lambda_{G'}(e) = \lambda_G(e)$ if e is not a loop (i.e. $e \in E_G$) and $\lambda_{G'}(e) = \ell$ if $e = f(v, \ell)$ for some $v \in V_G, \ell \in \kappa_G(v)$, i.e. if e is one of the newly added loops.

Summary. Formally we use the edge-labeled multigraphs of Definition 2.2, with node labels encoded as loops. Where this formal precision is not necessary, e.g. in examples, we still write node labels inside the node, and use the simpler language of Definition 2.1. There is a straightforward formal relationship to the definition of graphs in C&E (here Definition 2.3), see Remark 2.4. This is important since much of this thesis is based on the HR algebra of C&E (see Section 2.4).

We can now establish some further foundational concepts related to graphs.

Definition 2.5 (Subgraphs)**.** Let G and H be graphs. We say that H is a *subgraph* of G,written $H \subseteq G$, if $V_H \subseteq V_G$,$E_H \subseteq E_G$, and both $vert_H(e) = vert_G(e)$ and $\lambda_H(e) = \lambda_G(e)$ for all $e \in E_H$.
The subgraph of G *induced* by a vertex set $U \subseteq V_G$ is the graph

$$H = \left(U, E_H, vert_G|_{E_H}, \lambda_G|_{E_H}\right)$$

where

$$E_H = \{e \in E_G \mid u, v \in U \text{ where } (u, v) = vert_G(e)\}.$$

We write $G - U$ for the subgraph induced by $V_G \setminus U$. For a subset of edges $F \subseteq E_G$ we write $G - F$ for the graph $\left(V_G, E_G \setminus F, vert_G|_{E_G \setminus F}, \lambda_G|_{E_G \setminus F}\right)$. For brevity, if $v \in V_G$ and $e \in E_G$ are a single node and edge respectively, we sometimes write $G \setminus v$ for $G \setminus \{v\}$ and $G \setminus e$ for $G \setminus \{e\}$.

Another important concept in analyzing the structure of graphs is that of paths and connectivity.

Definition 2.6 (Paths and connectivity)**.** For two vertices $u, v \in V_G$, we write $u \overset{e}{\leftrightarrow} v$, or just $u \leftrightarrow v$ if there is an edge e such that $vert_G\left(e\right) = (u, v)$ or $vert_G\left(e\right) = (v, u)$, i.e. if there is an edge between the vertices no matter the direction. If we care about the direction, we write more specifically $u \overset{e}{\to} v$ or just $u \to v$ for the existence of an edge e such that $vert_G\left(e\right) = (u, v)$.
A *path* from a vertex $u \in V_G$ to a vertex $v \in V_G$ is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_0 = u$, $v_k = v$ and $v_0 \overset{e_1}{\leftrightarrow} v_1 \overset{e_2}{\leftrightarrow} \ldots \overset{e_k}{\leftrightarrow} v_k$ with all different edges, i.e. $e_i \neq e_j$ for $i \neq j$. If in fact $v_0 \overset{e_1}{\to} v_1 \overset{e_2}{\to} \ldots \overset{e_k}{\to} v_k$, then we call the path *directed*. Note that we allow the case $k = 0$, i.e. $u = v$, we call this a *trivial* path.
We say two vertices $u, v \in V_G$ are *connected* if there is a path from u to v. Note that this is a reflexive, symmetric and transitive relation, and thus divides the graph into equivalence classes called *connected components*. We call a graph connected if it has a single connected component, i.e. any two vertices are connected.

Often in this thesis, we will work with graphs *up to isomorphism*. The following definition, again from C&E, explains this concept.

Definition 2.7 (Isomorphism)**.** Let G and H be graphs. An *isomorphism* $h : G \to H$ is a pair of bijections (h_V, h_E) where $h_V : V_G \to V_H$ and $h_E : E_G \to E_H$, such that for every $e \in E_G$ with $(u, v) = vert_H(e)$, $vert_H(h_E(e)) = (h_V(u), h_V(v))$ holds. We denote by $G \simeq H$ the existence of an isomorphism $G \to H$ and say G and H are *isomorphic* (note that this is an equivalence relation, i.e. reflexive, symmetric and transitive).
By an *abstract graph*, we mean the equivalence class of a graph G, namely

$$[G]_{iso} = \{H \mid H \simeq G\}\,.$$

In contrast, we call G itself a *concrete graph.* We say that a concrete graph G is *isomorphic to an abstract graph* $[H]_{iso}$ if $G \simeq H$, i.e. G belongs to the isomorphism class $[H]_{iso}$; this is also written as $G \simeq [H]_{iso}$.

Thus, when we speak of graphs ‘up to isomorphism’, we mean abstract graphs. We talk about abstract graphs as we would about concrete graphs, except that we don’t specify exact objects for the vertices and edges. Figure 2.1(d) shows an abstract graph of which Figure 2.1(c) is a representative; simply the blue node and edge identities are dropped.

2.3 Abstract Meaning Representation

Abstract Meaning Representation (AMR) is a meaning representation in graph form. But what does a sentence mean? A common distinction is the one between *sentence meaning* and *speaker meaning*, where the former is the meaning inherent to the sentence itself, while the latter is the meaning the speaker intends to convey. When Mary says to Sue *The baby is cold*, then the sentence meaning is just the factual statement that the baby experiences coldness. However, in context, Sue may understand this as e.g. a request to close the window or turn up the heating. AMR focuses on sentence meaning, i.e. *semantics*.

For example, the sentence *The baby is cold* is represented by the AMR in Figure 2.2(a). The nodes in an AMR represent the core concepts in the sentence, and the edges their relations. The representations of the words are mostly simple – for the noun *baby*, just the label *baby* is used. For predicates, such as *cold*, senses are disambiguated using OntoNotes (Hovy et al. (2006); related to PropBank Kingsbury and Palmer (2002)). For example, the 'temperature' meaning of *cold* is *cold-01*, and disambiguated from *cold-02* (unfriendly, emotionless) and *cold-03* ((of a trail) no longer being fresh). OntoNotes also provides *argument roles* for predicates, such as the ARG1 edge label here. This denotes that *cold-01* has an ARG1 role, defined in OntoNotes as the thing being cold, and that *baby* has that role.

This allows AMR to encode the *who did what to whom* of a sentence. Take for example the AMR for *James loves Lily* in Figure 2.2(b). Here James takes the ARG0 role of the lover, whereas Lily takes the ARG1 role as the object of affection. To get a better understanding and intuition of the task of building AMR, let us have a look at some properties of these graphs.

Firstly, as the name suggests, AMR **abstracts** away from syntax. The sentences *James loves Lily*, *Lily is loved by James* and even *James's love for Lily* all have the same AMR in Figure 2.2(b).

The AMR in Figure 2.2(c), representing the word *baker*, illustrates how AMRs are **decompositional**, in the sense that they sometimes split the meaning of a word into separate parts, allowing for more generalization. Here, *baker* is represented as 'a person who bakes'. Another example is shown in Figure 2.2(d), where a *possession* is represented as 'a thing that is possessed'.

Further, each AMR has a *root*, denoted in this thesis by the bold outline on the node, that expresses the **focus** of the AMR. Usually this is the main predicate of

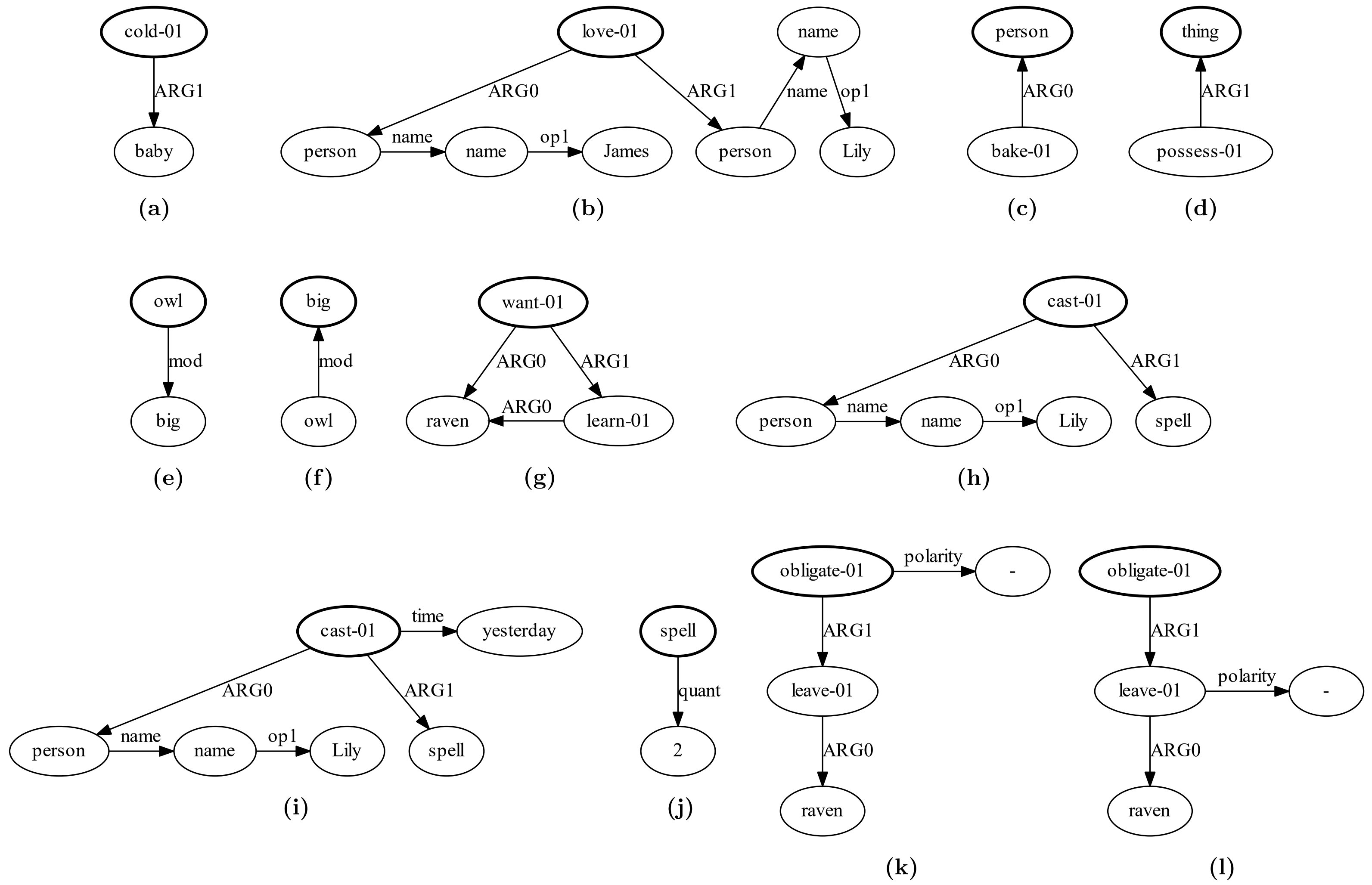


Figure 2.2: Several example AMRs.

the sentence.[1] The difference becomes clearer when looking at AMRs for fragments without predicate: compare the root placement for *the big owl* and *the owl is big* in Figures 2.2(e) and (f) respectively. These graphs are also examples for how AMR displays modification.

In the examples we saw so far, the AMRs were actually all trees. But AMRs can have more complex structure, with **reentrancies** as in the AMR for *The raven wants to learn* in Figure 2.2(g). Here, *raven* is both the wanter and the learner, and the additional ARG0 edge from *learn-01* to *raven* results in an (undirected) cycle, or reentrancy. In the **linear notation** for AMRs used e.g. in the published corpora, this graph would be written as `(w / want-01 :ARG0 (r / raven) :ARG1 (l / learn-01 :ARG0 r))`. This notation uses *node names* such as here `w`, `r` and `l` to identify nodes, allowing `(l / learn-01 :ARG0 r)` to unambiguously[2] refer back to the *raven* node as `r`. We will not use the linearized notation in this thesis, instead working on the graphs directly. In fact, it is sufficient to represent AMRs as **abstract graphs**, not specifying node names, since the node names are merely a tool to specify reentrancies.[3]

Further, AMR represents **no tense or aspect**, e.g. the graph in Figure 2.2(b) also represents the sentences *James loved Lily*, *James will love Lily*, and so on; similarly, the AMRs for *Lily casts spells* and *Lily is casting a spell* are identical as well (Figure 2.2(h)). Only when a specific time is mentioned in the sentence, then that time is represented in the AMR, see the graph for *Lily cast a spell yesterday* in Figure 2.2(i). In the same spirit, **grammatical number and definiteness is dropped** (*a spell*, *spells*, *the spell* and *the spells* are each just a *spell* node), but explicit numbers are represented; see Figure 2.2(j) for *two spells.* **Negation** is expressed through a '-' node and a polarity edge, and negation **scope** is somewhat represented by where the polarity edge attaches, note the difference between Figure 2.2(k) for *The raven doesn't have to leave* and Figure 2.2(l) for *The raven must not leave.* Quantifier scope is not encoded in AMR.

These simplifications make AMRs straightforward to read and understand, and made the annotation of large corpora possible. The latest release LDC2017T10[4] contains over 39,260 sentence-AMR pairs. These contain real life sentences from both newspaper and online discussion forums, that go far beyond the complexity of the examples discussed here. AMR has been experimented with in some sys-

[1] We will discuss how we represent the root formally at the end of Section 2.4.

[2] This is relevant when there are multiple nodes with the same label, e.g. here multiple *raven* nodes.

[3] In particular, the node names themselves are not taken into account by the official Smatch evaluation tool (Cai and Knight (2013)).

[4] `https://catalog.ldc.upenn.edu/LDC2017T10`

tems that take the graphs as input for an application, e.g. Wang et al. (2017) and Rao et al. (2017) for biomedical information extraction, and Dohare et al. (2018) for summarization. But mostly, AMR's 'general purpose' nature combined with a balanced combination of simplicity and challenging phenomena has made AMR a popular target representation for research in semantic parsing.

2.4 An algebra for building graphs

A core principle of this thesis is to build graphs step by step from smaller pieces. This section introduces the *HR algebra* of Courcelle and Engelfriet (2012) (C&E), a collection of simple, yet powerful operations to build graphs. This section has two parts: first, we formalize how we talk about 'operations' by introducing *algebras* and *terms.* Then we define the HR algebra.

2.4.1 Signatures, Terms and Algebras

This section mostly introduces standard notation, as used e.g. in Comon (1997) and C&E. Several sections are taken verbatim from these sources.

A *signature* (or *ranked alphabet*) is a pair (Σ, ar) where Σ is a finite set and ar is a mapping from Σ into the natural numbers $\mathbb{N}$. The arity of a symbol $f \in \Sigma$ is $\mathsf{ar}(f)$. The set of symbols of arity n is denoted by Σ_n. Elements of arity $0, 1, \ldots n$ are respectively called constant, unary, ..., n-ary symbols.

An *algebra* gives a signature meaning. An algebra $\mathcal{A}$ consists of a signature Σ and a set of objects $\mathcal{D}$ called the *domain*, and interprets each n-ary symbol f in Σ as an n-place function $f_{\mathcal{A}} : \mathcal{D}^n \to \mathcal{D}$ on the domain. Note how constant symbols are interpreted as functions that do not take arguments, i.e. constants in $\mathcal{D}$. We also call the functions $f_{\mathcal{A}}$ *operations.* A simple example is a *string algebra*, where the signature contains the binary concatenation symbol $*$ and words as constants, e.g. the set of symbols could be

$$\{*, \mathsf{James}, \mathsf{loves}, \mathsf{Lily}\}\,.$$

The domain is the set of strings. The word symbols evaluate to the words themselves, e.g. $\mathsf{James}_{\mathcal{A}} = \mathit{James}$, and $*_{\mathcal{A}}$ concatenates the words. For example, $*_{\mathcal{A}}(\mathsf{loves}_{\mathcal{A}}, \mathsf{Lily}_{\mathcal{A}})$ yields the phrase *loves Lily* and

$$*_{\mathcal{A}}(\mathsf{James}_{\mathcal{A}}, *_{\mathcal{A}}(\mathsf{loves}_{\mathcal{A}}, \mathsf{Lily}_{\mathcal{A}})) \tag{2.1}$$

yields the sentence *James loves Lily.* We sometimes drop the subscript $\mathcal{A}$, using just

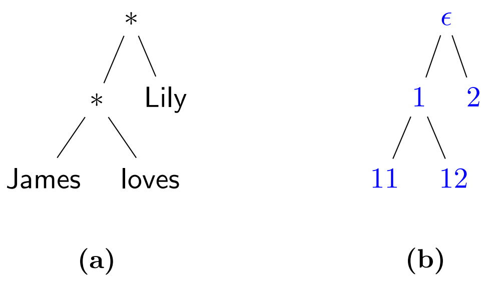


Figure 2.3: (a) A term t over the string algebra and (b) the positions in the term.

the symbol f to denote the operation $f_{\mathcal{A}}$ if this is clear from context. We also often use infix notation for binary operations such as $*$. That is, we sometimes simplify 2.1 to

$$\mathsf{James} * (\mathsf{loves} * \mathsf{Lily})\,.$$

A very useful notion is that of a *term*. Terms are (ordered) trees of symbols, that we can then evaluate in an algebra. Sometimes it will also be useful to have *variable* symbols in a term that function as placeholders. Formally, let $\mathcal{X}$ be an ordered set of constants called variables, usually we have $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. We assume that the sets $\mathcal{X}$ and Σ_0 are disjoint.

Definition 2.8 (Term)**.** The set $T(\Sigma, \mathcal{X})$ of terms over the ranked alphabet Σ and the set of variables $\mathcal{X}$ is the smallest set defined by:

(i) $\Sigma_0 \subseteq T(\Sigma, \mathcal{X})$ and

(ii) $\mathcal{X} \subseteq T(\Sigma, \mathcal{X})$ and

(iii) if $p \geq 1$, $f \in \Sigma_p$ and $t_1, \ldots, t_p \in T(\Sigma, \mathcal{X})$, then $f(t_1, \ldots, t_p) \in T(\Sigma, \mathcal{X})$.

If $\mathcal{X} = \emptyset$ then $T(\Sigma, \mathcal{X})$ is also written $T(\Sigma)$. Terms in $T(\Sigma)$ are called ground terms. A term t in $T(\Sigma, \mathcal{X})$ is *linear* if each variable occurs at most once in t.

Figure 2.3 shows a term over the string signature we just discussed. We can *evaluate* a term t in an algebra $\mathcal{A}$ by interpreting each symbol with the operation $\mathcal{A}$ assigns to it, and we write $[\![t]\!]_{\mathcal{A}}$ for the result of evaluating the operations bottom-up. Formally, for $t = f(t_1, \ldots, t_n)$, we can define this recursively as

$$[\![t]\!]_{\mathcal{A}} = f_{\mathcal{A}}\left([\![t_1]\!]_{\mathcal{A}}, \ldots, [\![t_n]\!]_{\mathcal{A}}\right).$$

For example, let t be the term in Figure 2.3 and $\mathcal{A}$ the string algebra from earlier, then

$$[\![t]\!]_{\mathcal{A}} = \textit{James loves Lily.}$$

If the algebra is clear from the context, we sometimes just write $[\![t]\!]$.

The *depth* (or *height*) of a term is again defined recursively. If a term t consists of just a constant, then t has depth 0. Otherwise, t has the maximum depth of its child terms plus one, i.e. if $t = f(t_1, \ldots, t_n)$ for $n > 0$, then the depth of t is

$$\max_{i=1,\ldots,n} d_i + 1$$

where d_i is the depth of t_i.

Sometimes it will be useful to talk about *where* exactly in a term a symbol occurs. To this end, we can also define a term $t \in T(\Sigma, \mathcal{X})$ as a partial function $t : \mathbb{N}^* \to \Sigma \cup \mathcal{X}$ with domain $\mathsf{pos}(t)$ satisfying the following properties:

(i) $\mathsf{pos}(t)$ is nonempty and prefix-closed.

(ii) $\forall p \in \mathsf{pos}(t)$, if $t(p) \in \Sigma_n$, $n \geq 1$, then $\{j \mid pj \in \mathsf{pos}(t)\} = \{1, \ldots, n\}$.

(iii) $\forall p \in \mathsf{pos}(t)$, if $t(p) \in \mathcal{X} \cup \Sigma_0$, then $\{j \mid pj \in \mathsf{pos}(t)\} = \emptyset$.

In other words, $\mathsf{pos}(t)$ is a set of *positions* or 'addresses', and in this view, t assigns a symbol from the signature (or a variable) to each position in $\mathsf{pos}(t)$. Conditions (ii) and (iii) ensure that arities are respected. Figure 2.3(b) shows example positions.

Essentially, the purpose of using variables in a term is so that we can replace them. We formalize this notion here.

Definition 2.9 (Context)**.** Let $\mathcal{X}_n = \{x_1, \ldots, x_n\}$ be an ordered set of n variables. A linear term $C \in T(\Sigma, \mathcal{X}_n)$ is called a context and the expression $C[t_1, \ldots, tn]$ for $t_1, \ldots, t_n \in T(\Sigma)$ denotes the term in $T(\Sigma)$ obtained from C by replacing variable x_i by t_i for each $1 \leq i \leq n$. We denote by $\mathcal{C}_n(\Sigma)$ the set of contexts over $(x_1, \ldots, x_n)$ and by $\mathcal{C}(\Sigma)$ the set of contexts containing a single variable (i.e. $\mathcal{C}(\Sigma) = \mathcal{C}_1(\Sigma)$).

With this background established, we can now have a look at the graph operations of the HR algebra.

2.4.2 The HR Algebra

In this section, we discuss the *HR algebra* of C&E, an algebra of simple and general graph combining operations.[5]

[5]The name HR algebra stems from the algebra's close relation to *Hyperedge Replacement Grammar* (HRG, Drewes et al. (1997)); this relation is throughly examined in C&E. We will not discuss

The definitions in this section are based on C&E (Definitions 2.24 to 2.32), but also contain simplifications and minor changes for the context of this thesis. In particular, the definitions are adapted to edge-labeled directed multigraphs.

At the core of the HR algebra are *s-graphs*, which are graphs with special, additional node labels called *sources*. Generally, we use capital letters for sources, printed red in figures. For example, in G_1 in Figure 2.4, the node 1 has an A source and node 2 has a B source. These sources mark the nodes where the HR algebra can 'glue' graphs together, as we will see below. The formal definition of s-graphs is as follows.

Definition 2.10 (Graphs with sources)**.** Let $\mathcal{S}$ be a fixed set. A *concrete graph with sources*, or a *concrete s-graph* for short, is a pair $G = \langle G^\circ, \phi_G \rangle$ consisting of a concrete edge-labeled directed mutigraph G° and a partial injective function $\phi_G : V_{G^\circ} \to \mathcal{S}$. We will simplify the notation V_{G° into V_G, and similarly for other notations. The domain of ϕ_G is a set of vertices denoted by $\mathit{Src}\,(G)$ and called the *set of sources* of G. If $\phi_G(u) = a$, then we say that u is the *a-source* of G.

In Figure 2.4 we have for example $\phi_{G_1} = \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{B}\}$. In G_8, node 1 has no source, i.e. ϕ_{G_8} is partial.

Before we proceed, let us transfer some terminology we established for graphs to s-graphs. We extend the notion of an isomorphism (Definiton 2.7) to concrete s-graphs in the obvious way, i.e. two concrete s-graphs G and H are isomorphic, $G \simeq H$, if $G^\circ \simeq H^\circ$ and the corresponding bijection $h_V : V_G \to V_H$ maps $\mathit{Src}\,(G)$ to $\mathit{Src}\,(H)$ and that $\phi_H(h_V(u)) = \phi_G(u)$ for all $u \in \mathit{Src}\,(G)$. Similarly, we obtain the notion of an *abstract s-graph* as the equivalence class of concrete s-graphs. The HR algebra operates on abstract s-graphs.

We say that an s-graph H is a subgraph of an s-graph G, if $H^\circ \subseteq G^\circ$ and the source assignments are identical, i.e. $\phi_H = \phi_G|_{V_H}$. But importantly, we say H is a *sub-s-graph* of G if just $H^\circ \subseteq G^\circ$, no matter what source assignments H has. These sub-s-graphs will be crucial in analyzing the compositional structure of a graph.

We can now define the merge operation, the key operation of the HR algebra that 'glues' graphs together. We first define it on concrete s-graphs, as an aide for defining it on abstract s-graphs later.

Definition 2.11 (Merge of concrete s-graphs)**.** Let G, H be concrete s-graphs. We say that H is a *subgraph of* G, written $H \subseteq G$, if $H^\circ \subseteq G^\circ$ and $\phi_H = \phi_G|_{V_H}$.

technical details of HRG in this thesis, but will look at related work based on HRG in Chapter 3.

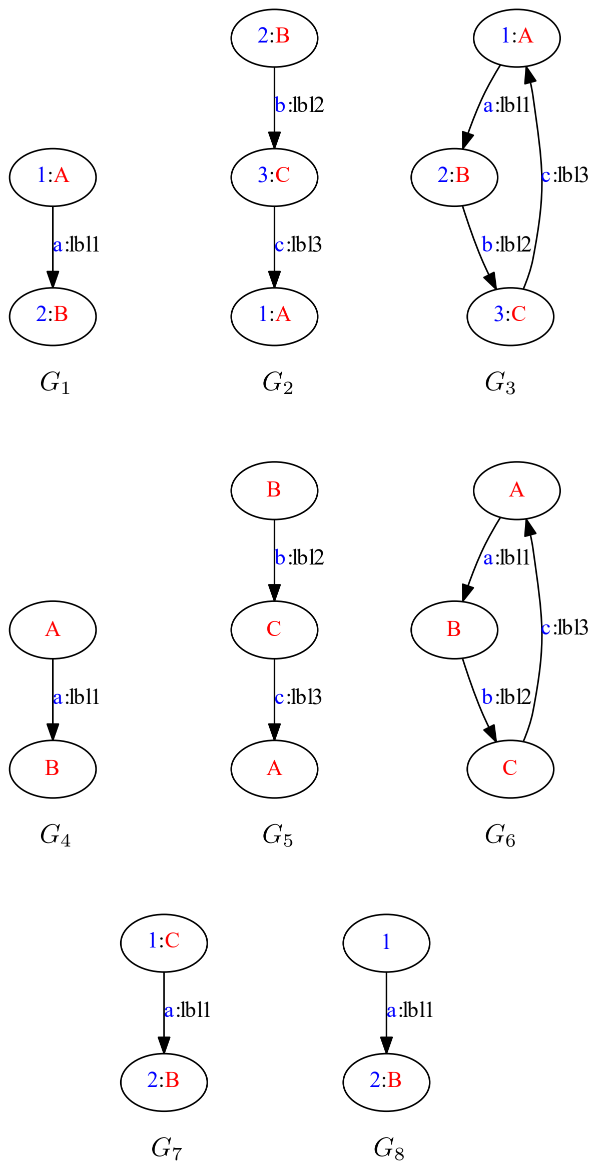


Figure 2.4: Example graphs.

Let G, H, K be concrete s-graphs. We write $G = H \mid\mid K$, and we say that G is the result of *merging* H and K, if and only if:

1. $H \subseteq G$, $K \subseteq G$, $G^\circ = H^\circ \cup K^\circ$,
2. $V_H \cap V_K = \mathit{Src}\,(H) \cap \mathit{Src}\,(K)$
3. $E_H \cap E_K = \emptyset$
4. $\phi_G = \phi_H \cup \phi_K$.

The last condition implies that ϕ_H and ϕ_K agree. The intuition behind this definition is that G is essentially just the union of the graphs, with the conditions that the graphs must overlap exactly where they have common sources, and everything except vertices with common sources has to come from exactly one of the graphs. In Figure 2.4, we have $G_3 = G_1 \mid\mid G_2$.

The merge for concrete s-graphs itself is of not much use for graph construction. On the one hand, it is obvious from the definition that there are pairs of concrete s-graphs that cannot be merged, i.e. that '$\mid\mid$' is only a partial function on concrete s-graphs. But more importantly, the idea of the merge is to 'glue' graphs together along their source names. But for concrete s-graphs, the nodes with the same source names, i.e. the nodes that will be 'glued' together, must be the same nodes in the first place. This pretty much defeats the purpose of marking them with an extra source name. But for abstract s-graphs, where graphs are only defined up to isomorphism and thus node identities are not accessible, the merge operation exactly matches the intuition of "gluing graphs together along their common source names".

Definition 2.12 (Merge)**.** Let H and K be abstract s-graphs. The abstract s-graph $H \mid\mid K$, called the result of *merging* H and K, is the isomorphism class of $H' \mid\mid K'$ for any two concrete s-graphs H' and K' such that $H' \simeq H$, $K' \simeq K$ and $H' \mid\mid K'$ is defined. For any two pairs (H', K') satisfying these conditions, the resulting s-graphs $H' \mid\mid K'$ are isomorphic. Hence $H \mid\mid K$ is well defined for any two s-graphs H and K. Informally we will say that H and K are 'glued at their sources with the same name'.

C&E show that the merge operation is indeed defined for any pair of abstract s-graphs. In Figure 2.4, the graphs G_4, G_5, G_6 are abstract s-graphs, i.e. we do not specify their node or edge identities, only the labels. Here, we have $G_6 =$

$G_4 \mid\mid G_5$. Note that G_2, G_3 and G_1 can play the roles of H', K' and $H' \mid\mid K'$ from Definition 2.12, respectively.

The following operations allow us to manipulate sources of s-graphs, so that we can control how two s-graphs merge.

Definition 2.13 (Forget)**.** Let $B \subset \mathcal{S}$ be a finite set of source names and $h : \mathcal{S} \to \mathcal{S}$ be the partial injective function that is undefined on B and the identity everywhere else. Let G be a concrete s-graph. Then let $\mathit{fg}_B(G)$ be the s-graph H such that the base graphs are equal, $H^\circ = G^\circ$, and $\phi_H = h \circ \phi_G$, that is all sources in B are removed.

In Figure 2.4, we have $G_8 = \mathit{fg}_{\{\mathsf{A}\}}(G_1)$.

Definition 2.14 (Rename)**.** Let $h : \mathcal{S} \to \mathcal{S}$ be a permutation of source names that is the identity outside a final set. Let G be a concrete s-graph. Then let $\mathit{ren}_h(G)$ be the s-graph H such that $H^\circ = G^\circ$ and $\phi_H = h \circ \phi_G$.

Let $\{\mathsf{A} \leftrightarrow \mathsf{C}\}$ be a permutation that swaps A and C, and is the identity apart from that. In Figure 2.4, we then have $G_7 = \mathit{ren}_{\{\mathsf{A}\leftrightarrow\mathsf{C}\}}(G_1)$.

Both rename and forget commute with isomorphisms, and can therefore be applied to abstract s-graphs. These three types of operations form the basis of the HR algebra. If we add some constant symbols for abstract s-graphs, we obtain a full HR algebra.

Definition 2.15 (The HR algebra of s-graphs)**.** Let $\mathcal{S}$ be a set of source names and $\mathfrak{C}$ be a set of constant symbols, each denoting an abstract s-graph with sources in $\mathcal{S}$. Then let

$$\Sigma_{\mathcal{S},\mathfrak{C}} = \{\ \mid\mid\ , \mathit{fg}_B, \mathit{ren}_h, c \mid B \subseteq \mathcal{S} \text{ finite}, h : \mathcal{S} \to \mathcal{S} \text{ permutation}, c \in \mathfrak{C}\}\,.$$

Further let $D_\mathcal{S}$ be the set of all abstract s-graphs with sources in $\mathcal{S}$. Then the *HR algebra with sources in $\mathcal{S}$ and constants in $\mathfrak{C}$* is the algebra with domain D_{HR} and signature $\Sigma_{\mathcal{S},\mathfrak{C}}$.

C&E suggest the following types of 'atomic' constants and show that it is particularly well suited to describe a large variety of graphs.[6]

[6]Specifically, C&E show that if there are k sources in $\mathcal{S}$, the HR algebra with the atomic constants can describe all graphs of *tree width* $k - 1$.

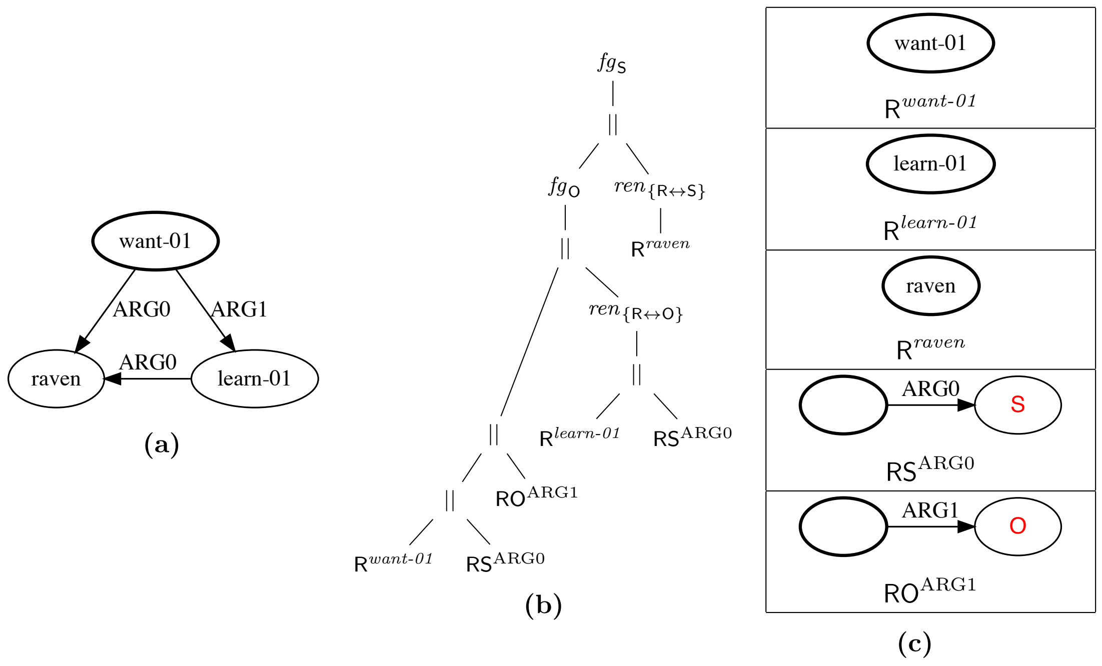
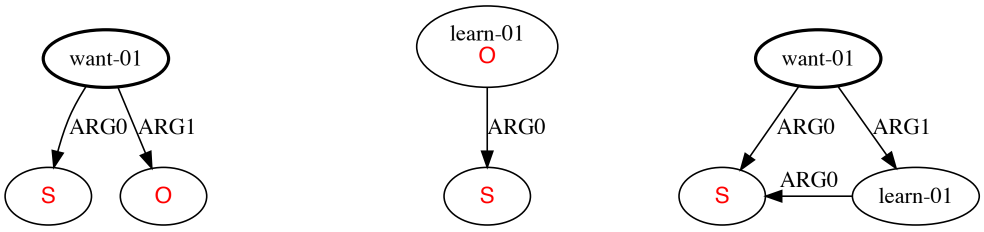


Figure 2.5: (a) An AMR for the sentence *The raven wants to learn*, (b) an HR term t that builds it and (c) the constants used in the term. For readability we write the node labels inside the nodes, even though we technically represent them as loops.

Definition 2.16 (Atomic s-graphs)**.** Let Λ be a set of labels as before and $\mathcal{S}$ a set of source names. For all $a, b \in \mathcal{S}$ and $\lambda \in \Lambda$ we define the following constant symbols. Let $\mathbf{a}$ be the constant symbol denoting the abstract s-graph consisting of a single node with an a source. Let $\mathbf{a}^{\ell}_{\lambda}$ denote the s-graph with a single node with an a source, at which there is a loop labeled λ. Finally, let $\mathbf{ab}_{\lambda}$ denote the s-graph with two nodes, one with an a source and one with a b source, and an edge from the former to the latter labeled λ.

C&E then use the constant set $\mathfrak{C} = \left\{\mathbf{a}, \mathbf{a}^{\ell}_{\lambda}, \mathbf{ab}_{\lambda} \mid a, b \in \mathcal{S}, \lambda \in \Lambda\right\}$ for some fixed set of labels Λ. We will deviate from that constant set in this thesis, and use constant sets more tailored to AMRs. We will define these constant sets in the respective chapters.

A side note on concrete graphs. We can also consider the partial *concrete* HR algebra, with domain over concrete s-graphs, constants that are concrete graphs and the other operations interpreted on the concrete graphs as defined above. As mentioned, this not very useful in most cases, since e.g. the merge operation on concrete graphs is only defined if nodes with the same source are the same node



(a) Result $[\![t_1]\!]$ of $t_1 = (\mathsf{R}^{\textit{want-01}} \mid\mid \mathsf{RS}^{\mathrm{ARG0}}) \mid\mid \mathsf{RO}^{\mathrm{ARG1}}$

(b) Result $[\![t_2]\!]$ of $t_2 = ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\,(\mathsf{R}^{\textit{learn-01}} \mid\mid \mathsf{RS}^{\mathrm{ARG0}})$

(c) Result $[\![t_3]\!]$ of $t_3 = fg_{\{\mathsf{O}\}}\,(t_1 \mid\mid t_2)$

Figure 2.6: Partial results for the term in Figure 2.5(b).

already before the merge, i.e. instead of freely merging nodes just based on sources, the node identities would need to be set up correctly in advance, defeating the purpose of the source names. In fact, the node identities of the constants in a term would persist all the way through evaluation. Interestingly though, in some technical contexts this property of being able to track nodes through the terms turns out to be useful. We will make use of it throughout Chapter 4 and in a proof in Chapter 5. But the intended graph combining power of the HR algebra shines when we use abstract graphs, which also fits with us modeling AMRs as abstract graphs.

The root in AMRs. In this thesis, we adapt the source system to feature a special source R, that we use to indicate the root of AMRs. We write this R source not in red, but instead mark the node where it is assigned with a bold outline. Apart from this notation, the source functions just as normal. For example, the graph in Figure 2.7 has the source assignment $\{1 \mapsto \mathsf{R}\}$.

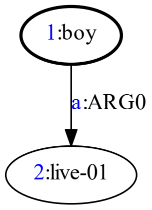


Figure 2.7: A concrete s-graph with a root source at node 1.

Example. Figure 2.5 shows in (a) the AMR for the sentence *The raven wants to learn* and in (b) an HR term t that evaluates to it; the atomic s-graphs are shown in (c).[7] Figure 2.6 shows partial results of the term. First, in the bottom left of the term, the *want-01* node $\mathsf{R}^{\textit{want-01}}$ is merged with the edges $\mathsf{RS}^{\mathrm{ARG0}}$ and $\mathsf{RO}^{\mathrm{ARG1}}$,

[7]Recall that while we technically represent node labels as loops, in examples we write the labels inside the node for readability, and that we indicate the root source R in examples with a bold node outline, instead of a red R.

attaching them at the R source. The result of this subterm

$$t_1 = \left(\mathsf{R}^{\textit{want-01}} \mid\mid \mathsf{RS}^{\text{ARG0}}\right) \mid\mid \mathsf{RO}^{\text{ARG1}}$$

is shown in Figure 2.6(a). In the sister term

$$t_2 = ren_{\mathsf{R}\leftrightarrow\mathsf{O}} \left(\mathsf{R}^{\textit{learn-01}} \mid\mid \mathsf{RS}^{\text{ARG0}}\right),$$

with the result shown in Figure 2.6(b), the *learn-01* node $\mathsf{R}^{\textit{learn}}$ is combined with the ARG0 edge $\mathsf{RS}^{\text{ARG0}}$, and its root R renamed to O. The term

$$t_3 = fg_{\{\mathsf{O}\}}\,(t_1 \mid\mid t_2)$$

combines these results, creating the graph in Figure 2.6(c). The *learn-01* node, now with the O source, gets plugged into the corresponding slot in the *want-01* graph, and the S sources merge, creating a joint subject slot. The O source is then forgotten since we no longer need it, with the result in Figure 2.6(c). In the full term in Figure 2.5, the *raven* node G_{raven} has its root R renamed to S to fit into that slot, we merge the graphs and forget the S source to obtain the final AMR.

This example highlights two important features of the HR algebra for semantic construction:

1. how the HR algebra allows us to create reentrancies through merging sources;[8] and

2. that we can use source names that are meaningful, i.e. R for root, S for subject and O for object.

In conclusion, the HR algebra is a powerful, flexible formalism to build graphs, with simple operations based on abstract graphs and source names.

2.5 Tree Automata

We have now seen the HR algebra and its graph building operations. But representing graphs through terms brings a challenge with it. For example, we might want to run an algorithm that takes as input the set of all terms that evaluate to a given graph. This could be to find the best term according to some scoring system, or to find patterns within the terms when learning a parsing model. But this set of

[8]Similar to *unification* in e.g. lexical-functional grammar (LFG; Kaplan et al. (1982)) and head-driven phrase structure grammar (HPSG; Pollard and Sag (1994)). More on that in Chapter 3.

terms is large – in fact, since the HR algebra's rename operation allows cycles, there are infinitely many terms to describe one graph. Besides that, working with sets of terms is cumbersome and inefficient. In Chapter 4 we will show how *tree automata* allow us to represent this infinite set of terms compactly (and in particular, finitely). This section establishes the background on tree automata.

Definition 2.17 (Tree Automaton)**.** A finite tree automaton over a signature Σ is a tuple $A = (Q, \Sigma, Q_f, \Delta)$ where Q is a set of states, $Q_f \in Q$ is a set of final states, and Δ is a set of transition rules of the following type:

$$f(q_1, \ldots, q_n) \to q,$$

where $n \geq 0$, $f \in \Sigma_n$ and $q, q_1, \ldots, q_n \in Q$. The state q is called the *parent*, and $q_1, \ldots, q_n$ the *children* (if $n = 0$, there are no children).

Tree automata over Σ run on ground terms over Σ. An automaton starts at the leaves and moves upward. Along such a run, the automaton inductively assigns a state to each subterm. If the direct subterms $u_1, \ldots, u_n$ of $t = f(u_1, \ldots, u_n)$ are labeled with states $q_1, \ldots, q_n$, then the term t will be labeled by some state q with $f(q_1, \ldots, q_n) \to q \in \Delta$. We now formally define the move relation defined by a tree automaton. Let $A = (Q, \Sigma, Q_f, \Delta)$ be a tree automaton over Σ. The move relation $\xrightarrow[A]{}$ is defined by: let $t, t' \in T(\Sigma \cup Q)$, then

$$t \xrightarrow[A]{} t' \Leftrightarrow \begin{cases} \exists C \in \mathcal{C}(\Sigma \cup Q), \exists f(q_1, \ldots, q_n) \to q \in \Delta, \\ \text{such that } t = C[f(q_1, \ldots, q_n)] \text{ and } t' = C[q]. \end{cases}$$

We write $\xrightarrow[A]{*}$ for the reflexive and transitive closure of $\xrightarrow[A]{}$.

A ground term t in $T(\Sigma)$ is *accepted* by a finite tree automaton $A = (Q, \Sigma, Q_f, \Delta)$ if

$$t \xrightarrow[A]{*} q$$

for some state q in Q_f. The *tree language* $L(A)$ recognized by A is the set of all ground terms accepted by A. A set L of ground terms is *recognizable* if $L = L(A)$ for some tree automaton A. Two tree automata are said to be *equivalent* if they recognize the same tree languages.

We can clarify the notion of how states participate in the process of how an automaton accepts a term. Let t be a ground term and A be a tree automaton, then a *run* r of A on t is a mapping $r : \mathsf{pos}(t) \to Q$ compatible with Δ, i.e. for every

$$\begin{aligned}
\mathsf{James} &\to [1,2] \\
\mathsf{loves} &\to [2,3] \\
\mathsf{Lily} &\to [3,4] \\
*\,([1,2],[2,3]) &\to [1,3] \\
*\,([2,3],[3,4]) &\to [2,4] \\
*\,([1,3],[3,4]) &\to [1,4] \\
*\,([1,2],[2,4]) &\to [1,4]
\end{aligned}$$

Figure 2.8: A decomposition automaton D for the sentence *James loves Lily* with respect to the string algebra. The states are pairs $[i, j]$ of string positions, denoting a span from i (inclusive) to j (exclusive) in the sentence. The final state is $[1, 4]$, covering the whole sentence.

position p in $\mathsf{pos}\,(t)$, if $t(p) = f \in \Sigma_n$, $r(p) = q$, $r(p_i) = q_i$ for each $i \in \{1, \ldots n\}$, then $f(q_1, \ldots, q_n) \to q$ must be in Δ. A run r of A on t is *successful* if $r(\epsilon)$ is a final state. Note that a ground term t is accepted by a tree automaton A if there is a successful run r of A on t.

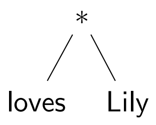
Further, we call a state $q \in Q$ *accessible* if there is a ground term $t \in T\,(\Sigma)$ such that $t \xrightarrow[A]{*} q$. We call a state $q \in Q$ *extensible* if there is a term $t \in T\,(\Sigma \cup Q)$ and a final state $q_f \in Q_f$ with $t \xrightarrow[A]{*} q_f$, such that q is a leaf of t. Note that a state $q \in Q$ can only participate in a successful run r, i.e. $q \in I\,(r)$, if q is both accessible and extensible. (However, not every accessible and extensible state q necessarily participates in a successful run, since the term used in proving the extensibility of q may have inaccessible states at its leaves).

2.5.1 Decomposition automata

While tree automata are a very general concept, one type of tree automaton is of particular importance here. For an algebra $\mathcal{A}$ over a signature Σ and an object c in its domain, we say a tree automaton D is a *decomposition automaton* for c with respect to $\mathcal{A}$, if its language $L\,(D)$ is exactly the set of all terms t in $T\,(\Sigma)$ that evaluate to d in $\mathcal{A}$, i.e. if

$$L\,(D) = \{t \in T\,(\Sigma) \mid [\![t]\!]_{\mathcal{A}} = d\}\,.$$

Take for example our sentence *James loves Lily* as d and the string algebra to decompose it. The automaton D in Figure 2.8 is a decomposition automaton for this scenario. Its states are of the form $[i, j]$, denoting a span from position i (inclusive) to j (exclusive). The first three rules denote at which positions the words occur. The other four rules denote all possibilities to concatenate two adjacent spans. For example, in the term



the automaton D assigns states $[2,3]$ and $[3,4]$ to the nodes loves and Lily respectively, and then uses the rule

$$*([2,3],[3,4]) \to [2,4]$$

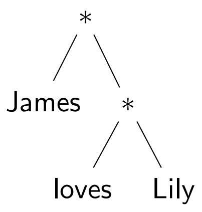
to assign the state $[2,4]$ to the full term. It can then use the rules

$$\mathsf{James} \to [1,2]$$

and

$$*([1,2],[2,4]) \to [1,4]$$

to assign the final state $[1,4]$ to the term



which indeed evaluates to our sentence.

Decomposition automata are useful as compact representations. For example, for a sentence of length n, there are $O(n^3)$ rules in a decomposition automaton like the one in Figure 2.8[9], whereas the number of terms is exponential in n.

2.5.2 Lazy automata

Sometimes, writing down the set of all rules in an automaton A can be undesirable, for example if the set of rules is very large, and not all of them need to be known to solve the given task. In other cases, just enumerating all rules may be difficult, since not all rules are known in advance. *Lazy* automata have their rule set not

[9]There is one constant rule per word in the sentence, i.e. $O(n)$ many, and the concatenation rules have the form $*([i,k],[k,j]) \to [i,j]$ with three variables i,j,k between 1 and n, yielding $O(n^3)$ rules.

explicitly given, but answer *queries* instead. There are two types of queries for a lazy representation of an automaton A.

Bottom-up: given states $q_1, \ldots, q_k$ and an algebra operation f, enumerate all the rules $q \to f(q_1, \ldots, q_k)$ in A. This asks how a list of given states can be combined into a new state q using the operation f.

Top-down: given a state q and an algebra operation f, enumerate all the rules $q \to f(q_1, \ldots, q_k)$ in A. This asks how a state can be derived from other states using the operation f.

For example, a string algebra decomposition automaton for a sentence of length n can be described in bottom-up queries in the following way: If the symbol f is a constant, return the rule

$$f \to [i, i+1]$$

for all i where the i-th word in the sentence is f. Further, if f is the concatenation '$*$', and the children are $[i, j]$ and $[k, l]$, then return the rule

$$*([i, j], [j, l]) \to [i, l]$$

if $j = k$, and no rule otherwise. This is a very compact representation of the automaton, and easily written as programming code.

There are algorithms that use lazy automata instead of explicit automata for efficiency, as we will see in Section 3.1. But we can also use these queries to enumerate all rules in the automaton A, turning a lazy representation into an explicit one. There is one algorithm each for bottom-up and top-down queries, which both use the queries to iteratively explore new states and rules. Since these algorithms play central roles in this thesis, we will look at them in some detail here, despite their simplicity.

A simple way to describe these algorithms is with a *deduction schema* consisting of *rules of inference* (Shieber et al. (1995)). The general form of a rule of inference as we use it here is

$$\frac{A_1, \ldots, A_k}{B_1, \ldots, B_k} \quad \langle \text{side conditions on } A_1, \ldots, A_k, B_1, \ldots, B_k \rangle,$$

where we diverge from Shieber et al. (1995) in that we have multiple items below the line. Here, $A_1, \ldots, A_k$ are the *antecedents* and $B_1, \ldots, B_k$ are the *consequents*. Essentially, if we know all antecedents $A_1, \ldots, A_k$ to be *derived* and the side conditions hold, the rule of inference states that we can also derive the consequents $B_1, \ldots, B_k$.

$$\frac{}{q} \quad \langle q \in Q_f \rangle$$

$$\frac{q}{q_1, \ldots, q_n} \quad \langle f\,(q_1, \ldots, q_n) \rightarrow q \in \Delta \rangle$$

Figure 2.9: Deduction schema for top-down exploration

$$\frac{q_1, \ldots, q_n}{q} \quad \langle f\,(q_1, \ldots, q_n) \rightarrow q \in \Delta \rangle$$

Figure 2.10: Deduction schema for bottom-up exploration

For example, in the top-down approach in Figure 2.9 the first rule initializes all final states as derived. The second rule states that for any rule $f\,(q_1, \ldots, q_n) \rightarrow q$ in the automaton, if we have the parent q derived, we obtain the children $q_1, \ldots, q_n$ as consequents. Since we only need q already derived and not $q_1, \ldots, q_n$, we can complete this deduction schema using only top-down queries. Whenever we successfully use the second of the two inference rules, we note the used rule as observed. Once all possible inferences have been made, all automata rules have been seen – or rather, all automata rules with extensible states, which is enough to get an automaton with the correct language. Algorithm 1 makes this process explicit. Line 1 initializes a set of seen (or derived) states and an agenda, as well as an empty rule set Δ. In the following **while** loop, derived states q are continuously popped from the agenda, and for each operation f, top-down queries for q and f are asked. If rules are found, they are added to the rule set and the child states $q_1, \ldots, q_n$ are, if new, added to the agenda and the set of seen states.

For the string decomposition automaton for *James loves Lily* in Figure 2.8, we would start with the final state $[1, 4]$, discovering first the rules

$$* \,([1, 3], [3, 4]) \rightarrow [1, 4],$$
$$* \,([1, 2], [2, 4]) \rightarrow [1, 4],$$

adding the states $[1, 3]$, $[3, 4]$, $[1, 2]$ and $[2, 4]$ to the agenda. Popping the states $[1, 3]$

Algorithm 1 Top-down exploration

```
1: seen := $Q_f$, agenda := $Q_f$ in arbitrary order, $\Delta := \emptyset$
2: while agenda $\neq \emptyset$ do
3:     pop state $q$ from agenda
4:     for $f \in \Sigma$ do
5:         for rule $f(q_1, \ldots, q_n) \to q$ given by top-down query do
6:             add $f(q_1, \ldots, q_n) \to q$ to $\Delta$
7:             for $i = 1, \ldots, n$ do
8:                 if $q_i \notin$ seen then
9:                     add $q_i$ to seen
10:                    add $q_i$ to agenda
11:                end if
12:            end for
13:        end for
14:    end for
15: end while
16: return $\Delta$
```

and $[2, 4]$ from the agenda yields the rules

$$*([1, 2], [2, 3]) \to [1, 3],$$
$$*([2, 3], [3, 4]) \to [2, 4]$$

respectively. This adds $[2, 3]$ to the agenda. The states $[1, 2]$ and $[3, 4]$ are there already, leaving the agenda with those three single word spans. Popping them gives us the last rules

$$\mathsf{James} \to [1, 2]$$
$$\mathsf{loves} \to [2, 3]$$
$$\mathsf{Lily} \to [3, 4].$$

Figure 2.10 shows a deduction schema for the algorithm that uses bottom-up queries. It has only one rule of inference: if all children of a rule are derived, we can also derive the parent. This same rule also initializes the process, for all constant symbols f where the list of children is empty. Since the child states are the antecedents, we can perform all deductions with this rule of inference using bottom-up queries only. This process finds all rules with reachable states – again enough to get an automaton with the correct language.

Algorithm 2 describes this process in practice. It has an extra feature in order to run efficiently: a lookup structure keeps track of all previously derived states, and can be queried with $\mathsf{lookup}\,(f, q', i)$ to return a list of derived states

$$\left(q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_{\mathsf{ar}(f)}\right)$$

that are candidates for a rule of the form

$$f\left(q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_{\mathsf{ar}(f)}\right) \to q$$

to exist. That is, assuming that a state q' is the i-th child in a query with symbol f, which previously derived states are good candidates for 'siblings' of q'. This type of indexing structure greatly increases efficiency, since not all possible combinations of derived states need to be tried whenever a new state is pulled.

Lines 1 and 2 initiate our structures, and Lines 3 to 12 initialize the agenda with all states obtained from constant rules, also noting those states as derived in seen and lookup. Then, while new states q' can be popped from the agenda, new applications of the inference rule in Figure 2.10 are tested by getting candidate siblings to q' from the lookup structure (Line 17) and asking bottom-up queries (Line 18). All such found rules are stored in Δ, and if the parent state q was not derived before, it is added to the agenda and stored as derived in seen and lookup.

We can see the importance of the lookup structure in string decomposition. Without it, i.e. if we were to try to concatenate any pair of derived spans, we would eventually try all span pairs $[i, j]$, $[k, l]$, even if j and k don't match and we cannot concatenate the spans. This yields a runtime of $O\left(n^4\right)$ – more than the $O\left(n^3\right)$ number of rules, because in the rules we always have $j = k$ guaranteed. But we can implement lookup to index derived spans by their start and end point, always returning matching spans only. That is, $\mathsf{lookup}\,(*, [i, j], 1)$ would only return spans starting on j so that $[i, j]$ can be the left sibling, and $\mathsf{lookup}\,(*, [i, j], 2)$ would return spans ending on i, so that $[i, j]$ can be the right sibling. This way, only concatenations $*\,([i, j], [k, l])$ are checked and the runtime goes down to the expected $O\left(n^3\right)$.

In our example automaton of Figure 2.8, we first obtain the states $[1, 2]$, $[2, 3]$ and $[3, 4]$ from the constant rules, putting them on the agenda and into lookup. If we then pull say $[1, 2]$, the query $\mathsf{lookup}\,(*, [1, 2], 2)$ returns no sibling q_1, since we have seen no state ending on 1. The query $\mathsf{lookup}\,(*, [1, 2], 1)$ returns as candidate sibling q_2 the state $[2, 3]$ starting at 2, but not $[3, 4]$. The query for children $[1, 2]$ and $[2, 3]$ with operation '$*$' then gives the rule

$$*\,([1, 2], [2, 3]) \to [1, 3].$$

We add the newly derived $[1, 3]$ to agenda and lookup and continue popping states until the agenda is empty.

The sort of indexing described here is standard (and trivial in the string algebra case), and often not even mentioned in the description of such algorithms. However, in later chapters we will need to use non-trivial lookup structures that are worth being mentioned and explained.

Algorithm 2 Bottom-up exploration

```
1: $\mathsf{seen} := \emptyset, \mathsf{agenda} := (), \Delta := \emptyset$
2: $\mathsf{lookup}$ := new lookup structure
3: for $c \in \Sigma, \mathsf{ar}(c) = 0$ do
4:     for rule $c \to q$ given by bottom-up query do
5:         add $c \to q$ to $\Delta$
6:         if $q \notin \mathsf{seen}$ then
7:             add $q$ to $\mathsf{seen}$
8:             add $q$ to $\mathsf{agenda}$
9:             add $q$ to $\mathsf{lookup}$
10:        end if
11:    end for
12: end for
13: while $\mathsf{agenda} \neq \emptyset$ do
14:    pop state $q'$ from $\mathsf{agenda}$
15:    for $f \in \Sigma, \mathsf{ar}(f) \geq 1$ do
16:        for $i = 1, \ldots, \mathsf{ar}(f)$ do
17:            for $\left(q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_{\mathsf{ar}(f)}\right) \in \mathsf{lookup}(f, q', i)$ do
18:                for rule $r = f\left(q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_{\mathsf{ar}(f)}\right) \to q$ given by
    bottom-up query do
19:                    add rule $r$ to $\Delta$
20:                    if $q \notin \mathsf{seen}$ then
21:                        add $q$ to $\mathsf{seen}$
22:                        add $q$ to $\mathsf{agenda}$
23:                        add $q$ to $\mathsf{lookup}$
24:                    end if
25:                end for
26:            end for
27:        end for
28:    end for
29: end while
30: return $\Delta$
```

3 Background: Semantic parsing

In the last chapter, we saw how we can build a graph from small, atomic units of meaning with a graph algebra. But we don't just want to build any graph, we want graphs that *represent the meaning of a sentence.* That is, we want to do *semantic parsing*, the process of computing a meaning representation for a given sentence.

We can define a *semantic parser* as an algorithm that takes a sentence as input and produces a semantic representation. In this thesis, that representation is an AMR. For clarification: while by *parser* we mean the fully implemented model, parameters and all, by *parsing formalism* we mean the formal framework that defines the mechanisms by which a parser can work. In this chapter, we discuss several AMR parsers and parsing formalisms. We highlight the challenges described in the introduction in more detail, and elaborate on the linguistic principles we will rely on in this thesis.

Section 3.1 introduces Interpreted Regular Tree Grammar (IRTG) as a parsing formalism that has a strong influence on this thesis. While its technical relation to this work is limited, the ideas IRTG illustrates reverberate throughout the thesis. Section 3.2 discusses the notion of compositionality, which gives a more general look on these ideas. On the basis of examples, and parallels to other parsing formalisms, we carve out the linguistic principles to guide our parser. Section 3.4 reviews related work in AMR parsing and Section 3.5 gives background on how parsers are induced and trained. Finally, Section 3.6 discusses this thesis' approach to semantic parsing.

automaton A_G	graph homomorphism h_g	string homomorphism h_s
$\mathsf{james} \to \mathrm{NP}$	G_{James}	James
$\mathsf{lily} \to \mathrm{NP}$	G_{Lily}	Lily
$\mathsf{love} \to \mathrm{VP}\,(\mathrm{NP})$	$fg_{\mathsf{O}}\,(G_{\mathrm{love}} \mid\mid ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\,(x_1))$	$\mathsf{loves} * x_1$
$\mathsf{apply\_subj}\,(\mathrm{NP}, \mathrm{VP}) \to \mathrm{S}$	$fg_{\mathsf{S}}\,(x_2 \mid\mid ren_{\mathsf{R}\leftrightarrow\mathsf{S}}\,(x_1))$	$x_1 * x_2$

Figure 3.1: A small handwritten IRTG that can analyze the sentence *James loves Lily*. The graph constants G_{X} are shown in Figure 3.2. The final state of the automaton A_G is S.

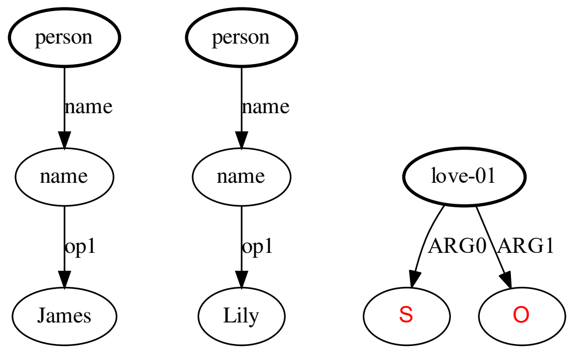
3.1 Interpreted Regular Tree Grammars (IRTGs)

This section introduces *Interpreted Regular Tree Grammars* (IRTGs, Koller and Kuhlmann (2011)) as an example formalism for AMR parsing. We mainly discuss IRTG here to illustrate the general linguistic principles that form the foundation for the AM algebra in Chapter 5 and our parser in Chapter 6. While there are many different formalisms illustrating these principles (a key point of Section 3.2), IRTG is convenient because it uses the technical tools we already introduced, namely algebras, the HR algebra in particular, and tree automata. IRTG also has a few technical applications in this thesis. Chapter 4 includes an application to graph parsing that involves IRTG, and a part of the parser in Chapter 6 can be phrased as an IRTG.

IRTG is a general framework for (potentially synchronous) grammars. It is implemented in Alto, available open source at `bitbucket.org/tclup/alto`; see also Gontrum et al. (2017).

Koller (2015) shows how to use a synchronous string-graph grammar based on the HR algebra for semantic construction. An example IRTG is shown in Figure 3.1. At its core is a set of grammar rules, represented here as a tree automaton (left column). Each rule is associated with a term over an algebra, here the HR algebra (center column) and the string algebra (right column) – we discuss details below. The flexibility of IRTG arises from the fact that in principle, any algebra can be used. For example, Koller and Kuhlmann (2012) introduce algebras that allow IRTG to model Tree Adjoining Grammar (TAG, Joshi and Schabes (1997)) with IRTG. Here, we focus on the combination of string and HR algebra shown in Figure 3.1, since this gives us a synchronous graph-string grammar, allowing us to translate strings to graphs.

We can see the rules in an IRTG as instantiations of general linguistic principles. We will highlight these principles in Section 3.2.



(a) G_{James} **(b)** G_{Lily} **(c)** G_{love}

Figure 3.2: The graphs referenced in Figure 3.1.

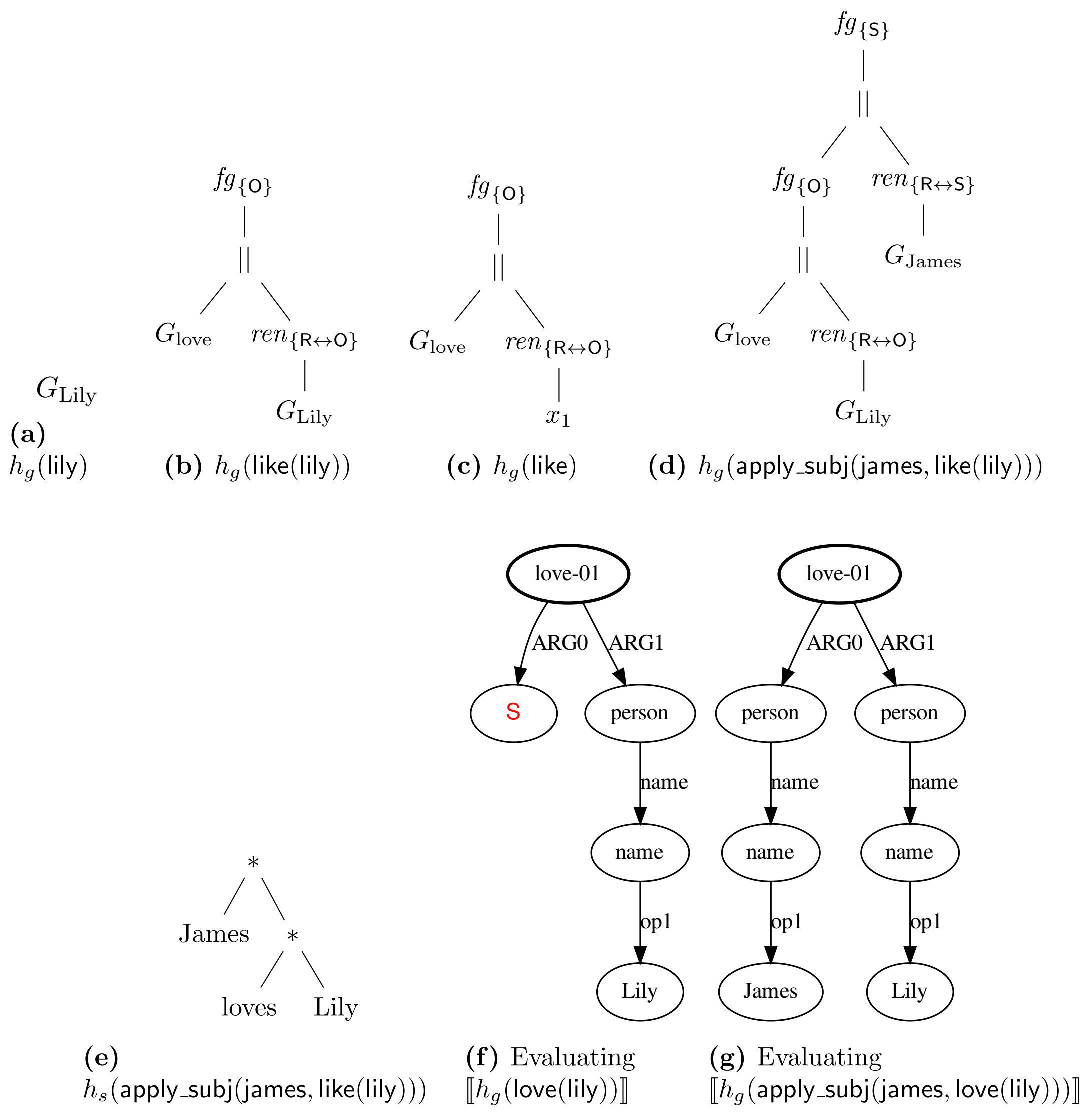
3.1.1 IRTG in detail

At the core of an IRTG is a set of grammar rules, here represented as a tree automaton A_G.[1] We call A_G the *grammar automaton*, the symbols in the signature of A_G the *rule labels*, and the trees in the language of A_G the *derivation trees*. The states of the grammar automaton, here NP, VP and S, play the role of *nonterminals* in a grammar, i.e. they restrict which derivation trees are allowed.

The second ingredient in an IRTG is *interpretations*. An interpretation consists of an algebra $\mathcal{A}$ and a *tree homomorphism* h that maps rule labels to contexts (i.e. terms with variables) over the signature of the algebra $\mathcal{A}$. The IRTG in Figure 3.1 has two interpretations, one to the HR algebra (second column) and one to the string algebra (third column).

We assume a homomorphism to map a rule label of arity k to a term with variables $x_1, \ldots, x_k$. For example, the homomorphism h_g maps the rule label james of arity 0 to the term G_{James} and maps the rule label $\mathsf{apply\_subj}$ of arity 2 to the term $fg_{\mathsf{S}}\left(x_2 \mid\mid ren_{\mathsf{R}\mapsto\mathsf{S}}\left(x_1\right)\right)$. Such homomorphisms can map trees to trees, by recursively plugging partial results into contexts bottom up; formally for a tree $t = f\left(t_1, \ldots, t_k\right)$ we have $h\left(t\right) = h\left(f\right)\left[h\left(t_1\right), \ldots h\left(t_k\right)\right]$, where the square brackets indicate that we replace each variable x_i with $h\left(t_i\right)$, as introduced in Section 2.4.1. For example, Figure 3.3(a) shows G_{Lily} as $h_g\left(\mathsf{lily}\right)$ and Figure 3.3(c) shows $h_g\left(\mathsf{love}\right)$. To get the homomorphic image $h_g\left(\mathsf{love}\left(\mathsf{lily}\right)\right)$ of the combined tree, we simply plug in G_{lily} into the x_1 slot of $h_g\left(\mathsf{love}\right)$, obtaining the HR term in Figure 3.3(b). This term evaluates to the s-graph in Figure 3.3(f): the root of G_{Lily} is renamed to O, this is merged with G_{love} and the O source is forgotten.

[1]Often, the rules are represented in the form of a *regular tree grammar (RTG)*, which are closely related to tree automata, see e.g. Comon (1997).

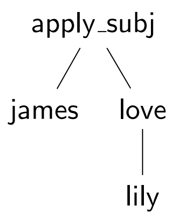


(a) $h_g(\mathsf{lily})$ **(b)** $h_g(\mathsf{like}(\mathsf{lily}))$ **(c)** $h_g(\mathsf{like})$ **(d)** $h_g(\mathsf{apply\_subj}(\mathsf{james}, \mathsf{like}(\mathsf{lily})))$

(e) $h_s(\mathsf{apply\_subj}(\mathsf{james}, \mathsf{like}(\mathsf{lily})))$ **(f)** Evaluating $[\![h_g(\mathsf{love}(\mathsf{lily}))]\!]$ **(g)** Evaluating $[\![h_g(\mathsf{apply\_subj}(\mathsf{james}, \mathsf{love}(\mathsf{lily})))]\!]$

Figure 3.3: Several homomorphic images of the IRTG in Figure 3.1.

Figure 3.3(d) shows the homomorphic image under h_g of the full derivation tree



we built in Figure 3.3. This HR term in Figure 3.3(d) evaluates to the AMR in (g). Similar to how G_{Lily} was fit into the O slot of G_{love}, now G_{James} is fit into the S slot. Its root R is renamed to S, the graphs merge and S is forgotten.

With a similar process, the other homomorphism h_s to the string algebra maps the derivation tree to the term in Figure 3.3(e). This term evaluates to the sentence *James loves Lily*, matching the AMR. Thus, our IRTG can generate a derivation tree that can be interpreted as generating both the sentence as well as the corresponding AMR.

The rules in Figure 3.1 all have a purpose. The first two rules james and lily simply introduce the respective constants as noun phrases. The third rule, love, is more interesting. It introduces the constants G_{love} and loves, and combines them with the object. On the string side, the object is simply concatenated to loves according to SVO word order. On the graph side, the object x_1 is filled into the O slot of G_{love}, as we have just seen: first, the root R of x_1 is renamed to O and then the graphs are merged, fitting the original root of x_1 into the O slot of G_{love}. Then, the O source is forgotten: it is not needed anymore and removing it allows re-using the O source elsewhere (if we were to build a more complex sentence). The fourth rule, apply_subj, combines a verb with its subject with the same mechanism as the previous rule, using the S source. All these mechanisms are overt, observable and understandable by a human looking at the grammar. In particular, the source names S and O match the subject and object roles.

3.1.2 Parsing.

If we invert the generation process on one side, we can do semantic parsing, as shown in Figure 3.4: Starting with the sentence (top left), we first obtain a term over the string algebra that evaluates to it. We then need to compute the *inverse homomorphism* h_s^{-1} of this term, to obtain a derivation tree (top right) that generates the sentence. At this point, we check whether the derivation tree is *grammatical*, i.e. whether it is accepted by the grammar automaton A_G. From there, we simply use

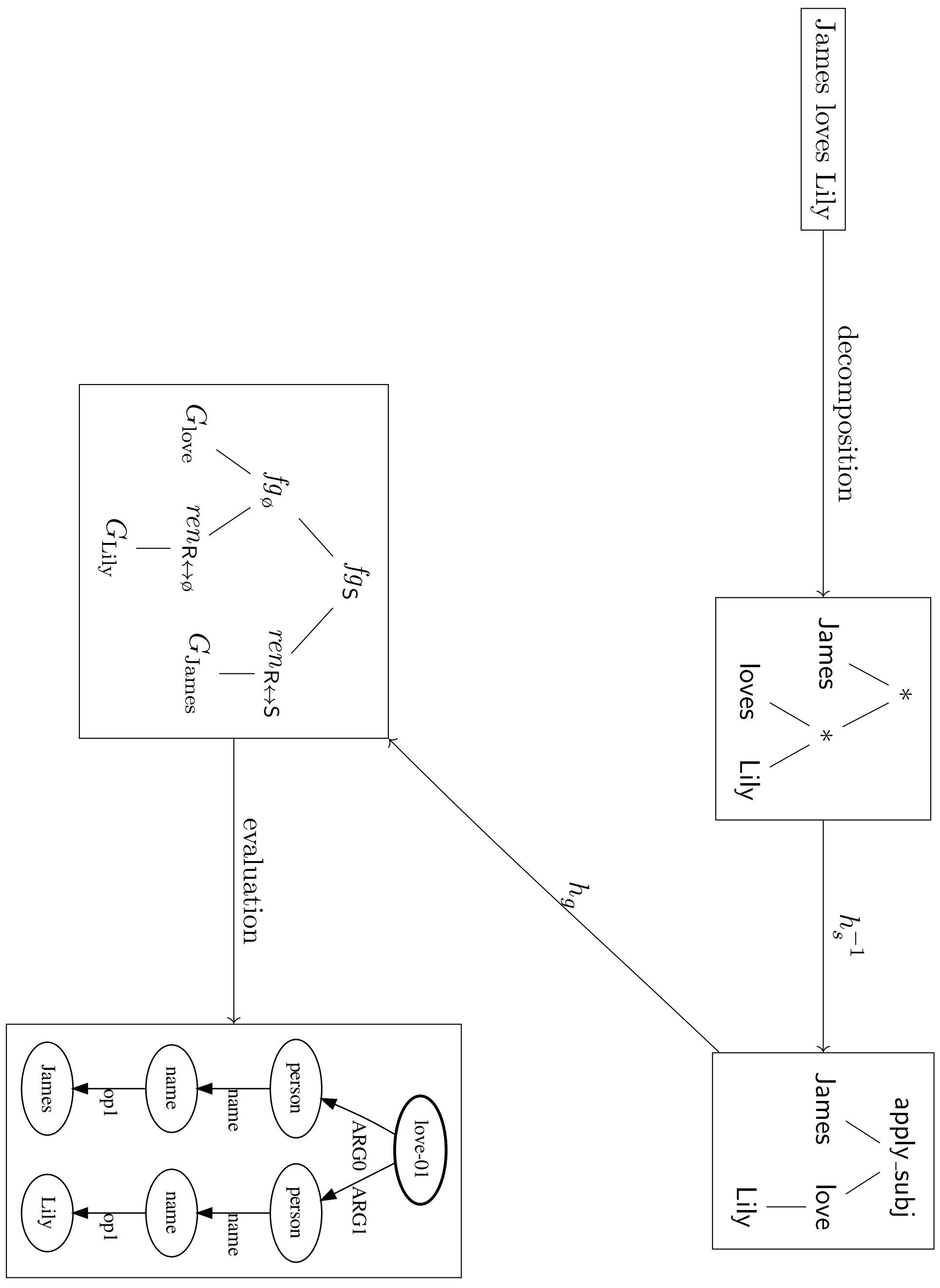


Figure 3.4: The semantic parsing process with an IRTG.

the homomorphism h_g to obtain an HR term (bottom left), that we can evaluate to the AMR (bottom right).

The beauty of this process is that all this can be computed efficiently with tree automata. For this, we represent all terms over the string algebra that evaluate to the input sentence in a decomposition automaton D. The set of all grammatical derivation trees that evaluate to the input then is

$$L(A_G) \cap h_s^{-1}(L(D)), \tag{3.1}$$

taking the inverse homomorphism h_s^{-1} of all valid algebra terms and intersecting with the grammars language to ensure grammaticality. These languages may be very large or even infinite, and thus expensive - or impossible - to compute. However, the languages of tree automata (which are the *regular tree languages*) are closed under inverse homomorphism and intersection. That is, there is an automaton I_D whose language is the inverse homomorphism of the language of D, i.e. $L(I_D) = h_s^{-1}(L(D))$. And, for any two tree automata B and C, there is an automaton $B \cap C$ whose language is the intersection of the languages of B and C, $L(B \cap C) = L(B) \cap L(C)$. Thus, we can write 3.1 as

$$L(A_G) \cap h_s^{-1}(L(D)) = L(A_G) \cap L(I) = L(A_G \cap I_D).$$

The automaton $A_G \cap I_D$ is also called the *parse chart*, because it is a generalization of conventional chart parsing. Groschwitz et al. (2016) present algorithms that compute the parse chart efficiently in practice, using lazy automata to avoid computing the decomposition and inverse homomorphism automata explicitly.

We can turn the IRTG into a statistical model by associating each rule of grammar automaton A_G with a weight. These weights propagate through intersection, so the parse chart $A_G \cap I_D$ is weighted and we can use the Viterbi algorithm to find the best derivation tree.

3.1.3 IRTG-related formalisms

A mechanism that is very closely related to IRTG with the HR algebra is *Hyperedge Replacement Grammar* (HRG; Drewes et al. (1997)), or to be precise, synchronous HRG (see e.g. Jones et al. (2012) and Peng et al. (2015)). Figure 3.5 shows an example derivation of (non-synchronous) HRG. The principle in HRG is that *hyperedges* labeled with nonterminals (in the example S,Y,...) are replaced by rules with graph fragments, that may themselves again contain nonterminals. Nodes on both the hyperedges and the replacement graphs in the rules are indexed (here in red) to

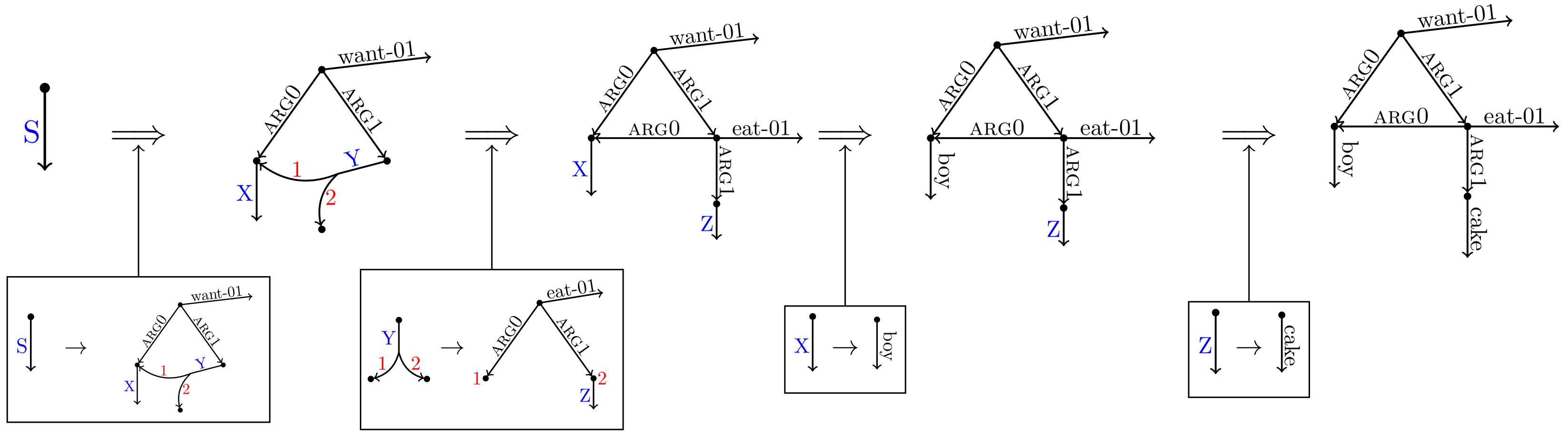


Figure 3.5: An HRG derivation for the AMR for *The boy wants to eat cake*; the rule for each derivation step is shown in the boxes.

indicate how exactly to insert the replacement graph. For example, the S edge is replaced with the *want* graph, that introduces nonterminals Y and Z. Subsequently, Y is replaces with a graph for *eat*, using the red indices to correctly attach the nodes; and so on. The derivation is finished when no nonterminals remain. When the hyperedge rules are paired with string rules, one obtains a synchronous grammar for strings and graphs, that can translate between sentences and AMRs.

A key difference between the HR algebra and HRG is that the HR algebra denotes the 'slots' along which graphs are combined with source names, whereas HRG uses indices.[2] While the source names of the HR algebra need extra management through rename and forget operations, they are more easily interpretable by human eyes and may help the model to generalize better.[3] Most importantly for this thesis, the source names will be crucial for the AM algebra of Chapter 5 and the parser in Chapter 6.

In conclusion, the promise of Koller (2015) is the following: a system for semantic construction that has transparent, interpretable rules that works efficiently in practice. However, in the paper this remains a promise, since Koller (2015) only describes the formalism, and leaves the problem of creating a working parser to future work. This promise is one of the starting points for this thesis.

3.2 Compositionality

IRTG is a *compositional* parsing formalism. We introduce compositionality in this section. Many formalisms for compositional semantic construction share similar methods, which we discuss on the base of examples. These methods form the linguistic principles we base our parsing model on.

So, what is compositionality? Compositionality ties together the syntax (i.e. the surface structure) and semantics (i.e. the meaning) of a sentence. Specifically, the principle of compositionality states (in the formulation of Kracht (2011)):

> The meaning of a complex expression is a function of the meanings of its parts and the mode of composition by which it has been obtained from these parts.

[2]This difference of using names versus indices as 'addresses' is a widespread phenomenon, discussed e.g. as the difference between attribute-value and positional-value notations for grammars (see e.g. Johnson (1987)).

[3]Specifically, Koller writes: "Copestake et al. (2001), in particular, conclude from their experiences from designing large-scale HPSG grammars that explicitly named arguments simplify the specification of the syntax-semantics interface and allow rules that generalize better over different syntactic structures."

There are many slightly varying formulations of this, for example Szabó (2017) formulates several variations, with the central one being:

> The meaning of a complex expression is determined by its structure and the meanings of its constituents.

Just as the exact formulations vary, the interpretations vary as well. I will use the following interpretation, that I believe to be in line with Gamut (1991). The idea is that the structure, or "mode of composition", of a sentence (or any natural language expression) is given by its syntax. Further, that the words (or more generally, the minimal constituents) have meanings assigned to them. The meaning of the complex expression can then be computed along the syntactic structure, where each operation of syntactic composition has a semantic composition function associated with it. That this is possible, determining the complex meaning only from the constituent meanings and the syntax structure, and moreover, that the structure of semantic composition follows the syntax, is the principle of compositionality.

What this also means is that a single notion of compositionality does not exist. Whether a language is compositional is only defined with respect to what syntax and what meaning representation we use. We can extend this idea to a semantic parser (or parsing formalism) if the parser defines a syntax. We can then examine whether the parser is compositional with respect to that syntax and with respect to the meaning representation it uses.

For example, the string-HR IRTG we discussed above is compositional with respect to the syntax defined by the derivation tree (related to the sentence via the string interpretation), and with respect to AMR s-graphs as meaning representations.

3.2.1 Compositional formalisms

Of course, IRTG is by far not the only formalism for compositional semantic construction. In fact, there is a long line of such formalisms – in particular the theory put forth by Montague in e.g. Montague (1973); but also Minimal Recursion Semantics (MRS, Copestake et al. (2005)) and Glue (Dalrymple et al. (1997)) for HPSG (Pollard and Sag (1994)) and LFG (Kaplan et al. (1982)); as well as Combinatory Categorial Grammar (CCG, Steedman (2000)).

We will not discuss all of these in detail, but instead look at methods and examples for compositional semantic construction through the eyes of IRTG. We then illustrate the point that these methods are shared across formalisms with a brief

automaton A_G	graph homomorphism h_g	string homomorphism h_s
james $\rightarrow$ NP	G_{James}	James
lily $\rightarrow$ NP	G_{Lily}	Lily
owl $\rightarrow$ N	G_{owl}	owl
raven $\rightarrow$ N	G_{raven}	raven
lion $\rightarrow$ N	G_{lion}	lion
snake $\rightarrow$ N	G_{snake}	snake
the (N) $\rightarrow$ NP	x_1	$\mathit{the} * x_1$
white (N) $\rightarrow$ N	$ren_{\mathsf{M}\leftrightarrow\mathsf{R}}\left(G_{\text{white}}\right) \Vert x_1$	white $* x_1$
learn $\rightarrow$ VP	G_{learn}	learn
leave $\rightarrow$ VP	G_{leave}	leave
twist $\rightarrow$ VP	G_{twist}	twists
shout $\rightarrow$ VP	G_{shout}	shouts
love (NP) $\rightarrow$ VP	$fg_{\mathsf{O}}\left(G_{\text{love}} \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\left(x_1\right)\right)$	loves $* x_1$
want (VP) $\rightarrow$ VP	$fg_{\mathsf{O}}\left(G_{\text{want}} \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\left(x_1\right)\right)$	wants $*$ (to $* x_1$)
persuade (NP, VP) $\rightarrow$ VP	$fg_{\mathsf{O}}\left(fg_{\mathsf{O2}}\left(G_{\text{persuade}} \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{O2},\mathsf{S}\leftrightarrow\mathsf{O}}\left(x_2\right)\right) \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\left(x_2\right)\right)$	persuades $* x_1 *$ to $* x_2$
apply_subj (NP, VP) $\rightarrow$ S	$fg_{\mathsf{S}}\left(x_2 \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{S}}\left(x_1\right)\right)$	$x_1 * x_2.$
and_VP (VP, VP) $\rightarrow$ VP	$fg_{\mathsf{op1,op2}}\left(\left(G_{\text{and}} \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{op1}}\left(x_1\right)\right) \Vert ren_{\mathsf{R}\leftrightarrow\mathsf{op2}}\left(x_2\right)\right)$	$x_1 *$ and $* x_2.$

Figure 3.6: A larger handwritten IRTG that captures a range of the examples in Section 3.2.1. The graph constants G_{X} are shown in Figure 3.7

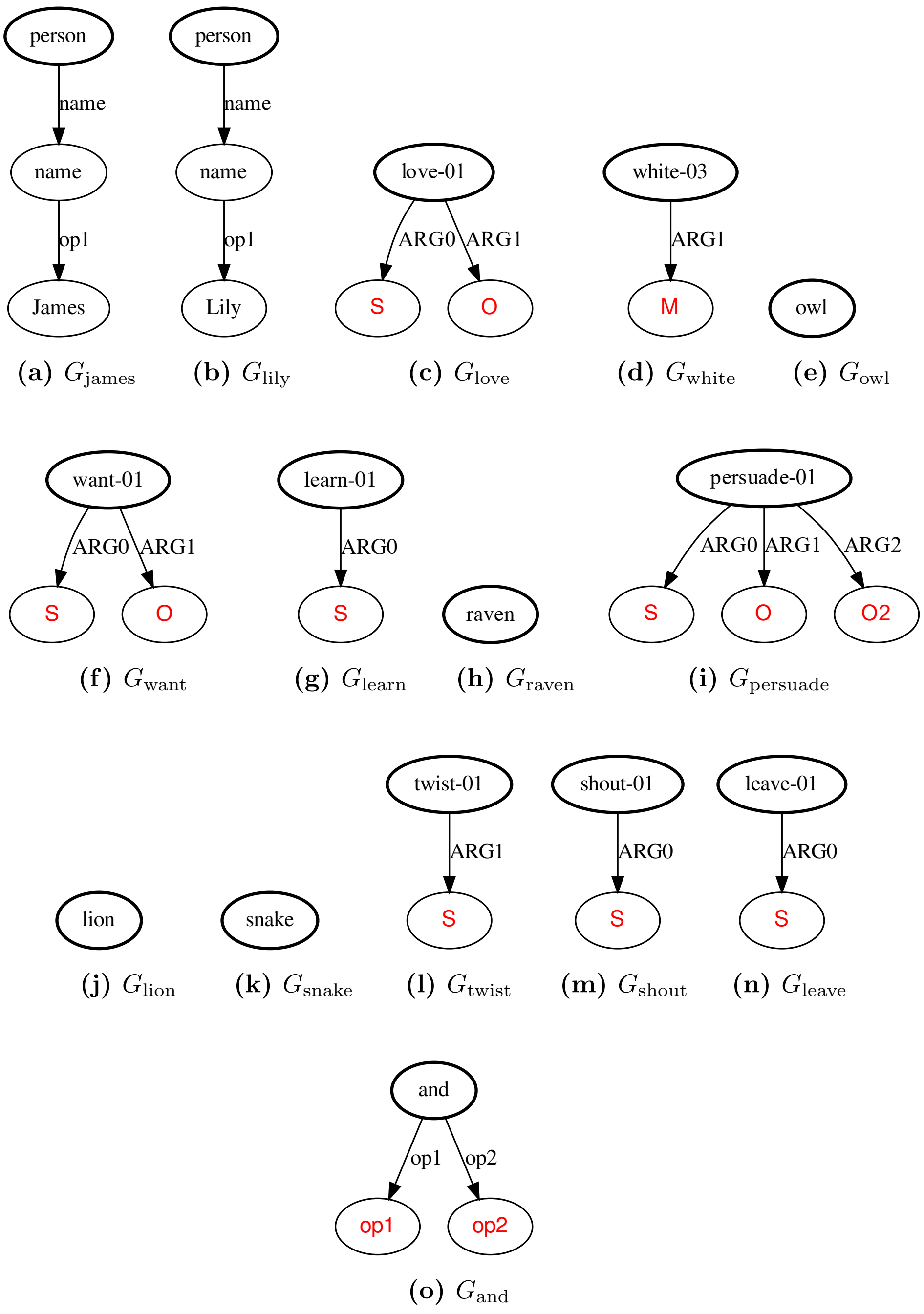
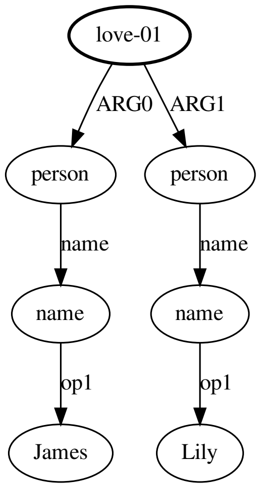
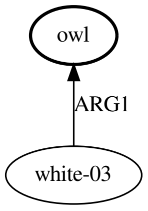


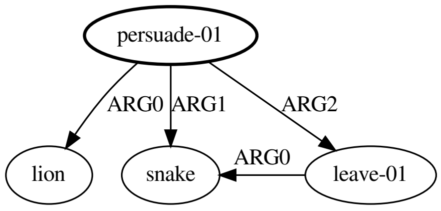
Figure 3.7: The graph constants used in the IRTG in Figure 3.6



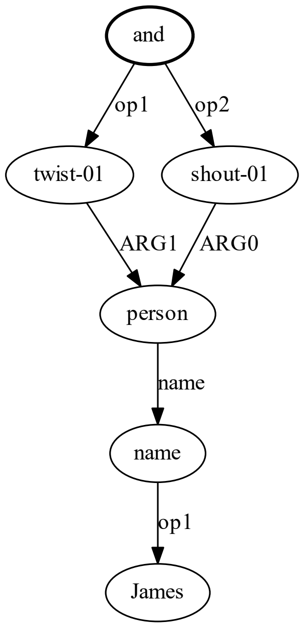
(a) James loves Lily



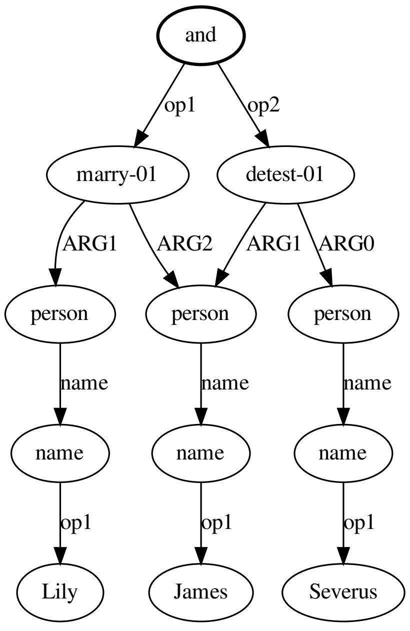
(b) A white owl



(c) The lion persuaded the snake to leave



(d) James twists and shouts



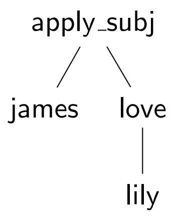
(e) Lily married and Severus detests James

Figure 3.8: Example AMRs

introduction to CCG.

3.2.2 Application, modification and types

For all following examples, we use the extended IRTG in Figure 3.6. Let us first discuss the mechanism of *application*. Consider again the sentence *James loves Lily*, and its AMR in Figure 3.8(a). We just saw how the AMR can be built with IRTG, using the derivation tree

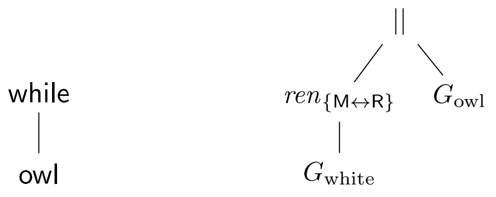


The relevant rules also exist in the IRTG in Figure 3.6. Now consider the s-graph constants in Figure 3.7, in particular the graph G_{love}. This graph, used in the IRTG approach, has two open sources S and O, indicating the two arguments subject and object. We call these nodes the 'argument slots'. We just saw in Section 3.1.1 how IRTG fills these slots. For example, in the love rule, the argument (here G_{Lily}) is inserted into the O slot with a sequence of rename, merge and forget operations of the HR algebra. We call this 'filling of slots' *argument application*. The rule apply_subj then applies the subject, by filling the S slot with G_{James}, as seen above. In this scenario, we call G_{love} the *head* and G_{Lily} and G_{James} the *arguments*.

Generally speaking, the left child of the apply_subj rule, the argument, has a root source R, and the right child, the head, has an S source; then the apply_subj rule fills that S slot of the head with the root of the argument. Note that this operation only makes sense if the argument indeed has an R source, and the head an S source. But then, at parsing time when we build a semantic term, how do we guarantee that these conditions are met?

The grammar can guarantee this by connecting *syntactic categories* with *semantic types*. We understand a semantic type here loosely as a summary of the unfilled argument slots a semantic representation has. For s-graphs,we can e.g. use the set of source names.[4] The IRTG here uses syntactic categories like NP or VP as nonterminals in the grammar automaton. The idea is to correlate the semantic types and syntactic categories: for example, the syntactic category VP corresponds to having exactly an S and an R source, and NP corresponds to having just a root R. When

[4]Courcelle and Engelfriet (2012) too call the set of sources in an s-graph its type, although in a different context.



(a) Derivation Tree **(b)** HR Term

Figure 3.9: IRTG derivation tree with HR term for *the white owl*.

parsing then, the syntactic categories restrict the set of possible derivation trees to ones that produce well-typed terms, i.e. terms where all operations obtain a meaning representation of adequate type.

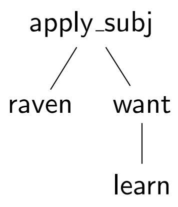
The application mechanism is complemented by *modification*. Take for example the sentence *the white owl*, with the AMR in Figure 3.8(b). A derivation tree and HR term for this sentence can be found in Figure 3.9. The rule white takes an N argument with just an R source, and the result also has category N as well as just an R source. In particular, modification does not change the semantic type. This also means that while application uses up an argument slot of the head, modification can be repeated (take for example *the fast white owl*, *the beautiful fast white owl* and so on). Modification also occurs for other types than just nouns; for example, adverbs modify verb phrases (VP).

3.2.3 Unification

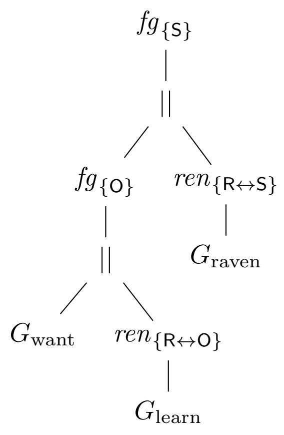
A more complex mechanism is what I will call in this thesis *unification*.[5] The phenomenon of **subject control** illustrates it well, as in *The raven wants to learn* with the AMR in Figure 3.8(c) (we already saw this example in the introduction). Here, something interesting happens: semantically, the raven is both the wanter and the learner. But *learn* has no overt subject in the sentence, the meaning of the raven being the learner is obtained implicitly through the control verb *want*.

How do compositional models handle this challenge? Figure 3.10 shows a derivation with the IRTG in Figure 3.6. Recall that when two graphs are merged in the HR algebra, nodes with equal source names are fused together. Thus, when the graphs G_{want} and G_{learn} are merged, their nodes with S sources get unified automatically; see Figure 3.10(c). Only then is G_{raven} merged into that joint subject slot with the apply_subj rule. Figure 3.10(a) shows the derivation tree, and (b) the corresponding HR term, where the lower merge operation combines G_{want} and G_{learn}, and the upper

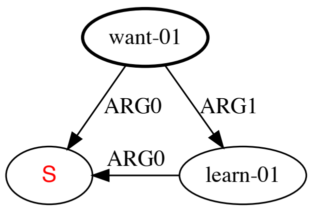
[5]I use the word in a broader sense as described in this section. I do not mean e.g. the technical notion of unification in feature structures of HPSG.



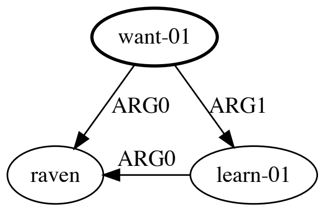
(a) Derivation tree



(b) HR term

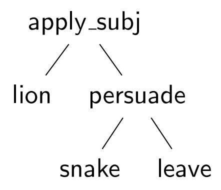


(c) Intermediate results of want(learn), i.e. $fg_{\{\mathsf{O}\}}(G_{\text{want}} \;||\; ren_{\{\mathsf{R}\leftrightarrow\mathsf{O}\}}(G_{\text{learn}})\;)$

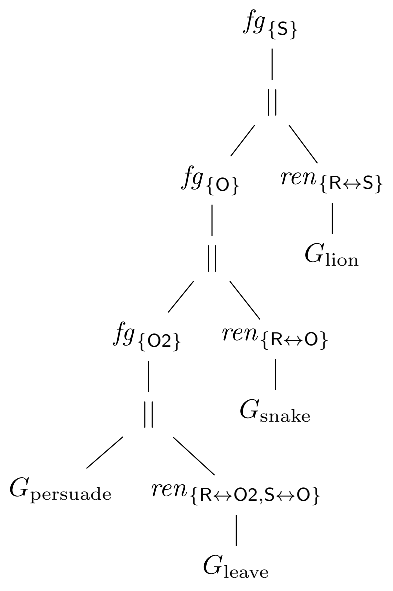


(d) Final AMR for *The raven wants to learn*

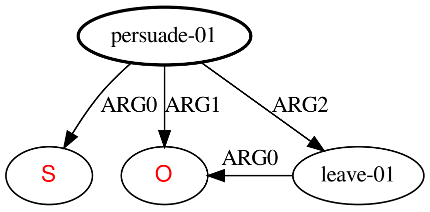
Figure 3.10: IRTG analysis of *The raven wants to learn.*



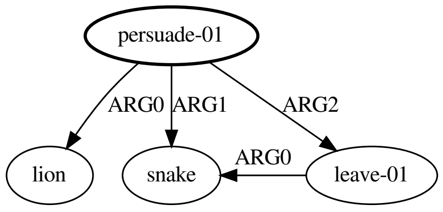
(a) Derivation tree



(b) HR term

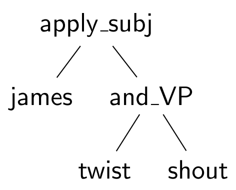


(c) Intermediate results of persuade(leave), i.e. $fg_{\{O2\}}\left(G_{\text{persuades}} \mid\mid ren_{\{R\leftrightarrow O2, S\leftrightarrow O\}}\left(G_{\text{leave}}\right)\right)$

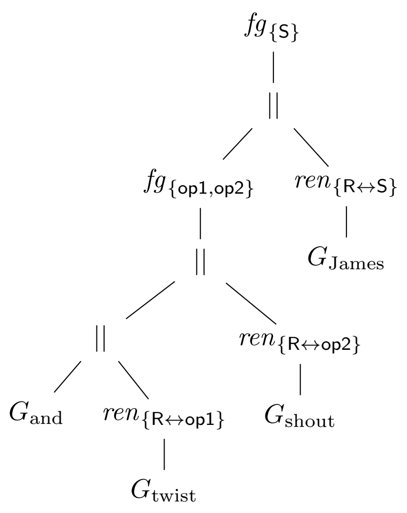


(d) Final AMR for *The lion persuades the snake to leave*

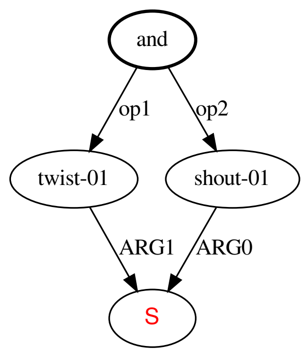
Figure 3.11: IRTG analysis of *The lion persuades the snake to leave.*



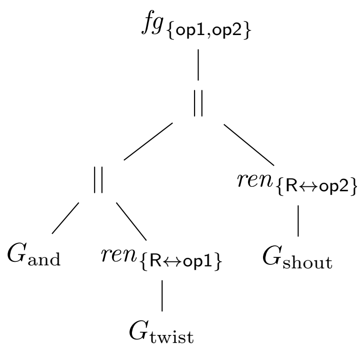
(a) Derivation tree


$fg_{\{S\}}$
$||$
$fg_{\{op1,op2\}}$
$ren_{\{R\leftrightarrow S\}}$
G_{James}
$||$
$||$
$ren_{\{R\leftrightarrow op2\}}$
G_{and}
$ren_{\{R\leftrightarrow op1\}}$
G_{shout}
G_{twist}

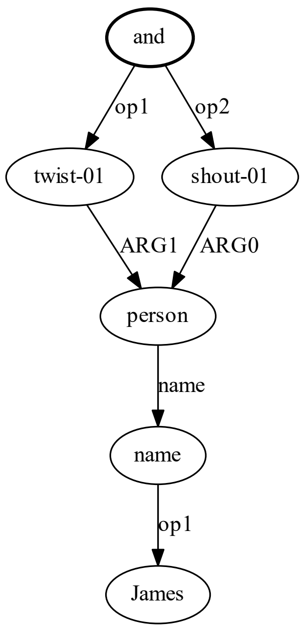
(b) HR term



(c) Intermediate results of and_VP(twist, shout), i.e. of the HR term in (d)


$fg_{\{op1,op2\}}$
$||$
$||$
$ren_{\{R\leftrightarrow op2\}}$
G_{and}
$ren_{\{R\leftrightarrow op1\}}$
G_{shout}
G_{twist}

(d) HR term for and_VP(twist, shout).



(e) Final AMR for *James twists and shouts*

Figure 3.12: IRTG analysis of *James twists and shouts.*

merge fits G_{raven} in the subject slot.

Note that this unification is encoded in the rule for want. The VP category of the object argument (here G_{learn}) guarantees that this argument still has an open S slot, as we saw in the discussion on types above. The merge operation in the want rule then performs the unification. We will see below that encoding the unification as a lexical property of a control verb like *want* is a common pattern in compositional formalisms.

A similar case to subject control is **object control**, as in

(1) the lion persuaded the snake to leave

where the subject of *leave* is now equal to the *object* of *persuaded*; see the AMR in Figure 3.8(d) and derivation in Figure 3.11. Note how the IRTG rule for *persuaded* achieves this by renaming the S source of *leave* to an O source with the $ren_{\mathsf{R}\mapsto\mathsf{O2},\mathsf{S}\mapsto\mathsf{O}}$ operation (see the parent to G_{leave} in Figure 3.11(b)). This yields the partial result in Figure 3.11(c), where afterwards G_{snake} and G_{lion} are fit into the O and S slots respectively, in the usual manner.

Coordination combines two phrases of the same syntactic category into one. In *James twists and shouts* (AMR in Figure 3.8(e); IRTG analysis in Figure 3.12), the phrase *twists and shouts* combines the intransitive verbs *twists* and *shouts*, and acts itself as if it were an intransitive verb again. The IRTG rule for *and* in Figure 3.6 reflects that, mapping two VP children to a VP result. Note in Figure 3.12(c) how the S sources of the arguments merge, creating the reentrancy in the AMR.

One complication arising with coordination is that phrases that are not usually considered constituents can be coordinated as well. For example in *Lily married and Severus detests James* (AMR in Figure 3.8(f)), a case of **right node raising**, the phrases *Lily married* and *Severus detests* are coordinated. Usually however, transitive verbs are combined with their object first, and only then with their subject. For example, the apply_subj rule in our IRTG expects a VP argument, i.e. a verb phrase only missing its subject, and the rule love immediately combines the intransitive verb with its object. Koller (2015) does not discuss right node raising. This is an example where the flexibility of the AM dependency approach of Chapter 6 will be particularly useful. Constraints on the syntax and on application order are relaxed there[6], and thus it is no problem that we need to combine non-standard constituents

[6]While this relaxation can lead to over-generation, i.e. to allowing constructions that would usually be considered ungrammatical, the subject of this thesis is not to develop a grammar for English, but to develop an AMR parser. For parsing, we rely on the neural model described in Chapter 6 to disambiguate between desired and undesired analyses. If the AM dependency approach were to be used in a grammar for English, additional restrictions would be necessary to prevent overgeneration.

or that we need to add arguments in an unusual order.

3.2.4 Excursion: CCG

To get a broader understanding of mechanisms and challenges in compositional semantic parsing, let us thus also consider CCG as a further representative. CCG has shown both theoretical and computational success (see e.g. Steedman (2000), Clark and Curran (2004), Lewis and Steedman (2014)), and has inspired much of the work in this thesis.[7] While CCG has no direct technical connection to this thesis, it illustrates the broader background of the methods we use.

CCG parses sentences into the semantic representation of *logical formula*. For example, the sentence *James loves Lily* has the logical formula $love'(james', lily')$, where $love'$ is a two place function and $james'$ and $lily'$ its arguments. The primes mark that e.g. $love'$ is a fixed semantic object, such as a function, whose details are of no immediate interest here.

Application and types. CCG build a logical formula by assigning to each word both a syntactic category and a semantic interpretation. For example, *James* and *Lily* would have syntactic category NP, paired with the semantic representations $james'$ and $lily'$ respectively. The verb *loves* however has a complex category $(S \setminus NP)/NP$ paired with the lambda term $\lambda y.\lambda x.love'\ (x, y)$. The category $(S \setminus NP)/NP$ denotes a function with two arguments of type NP that returns type S. The outer argument is expected to the right, denoted with the forward slash '/'. That is, when we combine the word *loves* with *Lily* to its right, we have the phrase *loves Lily* of category $S \setminus NP$. Simultaneously, $lily'$ is entered as an argument in the lambda expression, yielding $\lambda x.love'\ (x, lily')$ as the meaning of the phrase. Now in $S \setminus NP$, the backslash '\' indicates that the next argument NP is expected on the left, so with the same principle we can form the sentence *James loves Lily* with meaning $love'\ (james',\ lily')$. Loosely speaking, the semantic type of a lambda expression describes the number and type of its arguments, and CCG too has a close relation between the syntactic categories and semantic types.

Modification. The logical form of *the white owl* is $white'\ owl'$, where $white'$ is a one place function that returns an object of the same type as owl'. The CCG syntactic type for *white* is N/N, taking an N and returning an N. Thus, CCG too uses modification that leaves syntactic categories and semantic types unchanged.

[7] In fact, the related work Artzi et al. (2015) describes an approach to AMR parsing that is directly based on CCG. We will discuss it in Section 3.4.

Control and coordination. Consider again the sentence *The raven wants to learn.* CCG uses the semantic representation

$$\lambda p.\lambda y.want'\left(p\left(ana'y\right)\right)y$$

for the control verb *want*, where $ana'y$ represents an anaphor bound to y. Combining this with the logical form for *learn* to fill the variable p, we obtain

$$\lambda y.want'\left(learn'\left(ana'y\right)\right)y$$

That is, instead of unifying the argument slots into one, the lambda expression explicitly specifies that the argument of $learn'$ and the argument of $want'$ are to be filled with the same variable. This has the same effect as the unification described for the IRTG above, that the raven is both the learner and wanter in the end. Again, this information is encoded in the lexicon entry for *want.* Object control is modeled similarly in CCG.

CCG uses a special category *CONJ* for the coordinator *and* that combines like types. It combines the sequence

$$S \setminus NP \quad CONJ \quad S \setminus NP$$

into the single type $S \setminus NP$. CCG also ensures that the two verbs share the subject, returning for the phrase *twists and shouts* the interpretation

$$\lambda x.and'\left(twist'\ x\right)\left(shout'\ x\right)$$

(details of this process go beyond the scope of this thesis, and are given in e.g. Steedman (2000)). CCG uses the special operations *type raising* and *composition* to address right node raising, see Steedman (2000).

3.2.5 Additional phenomena

This sections presents some additional phenomena that often pose a challenge to compositional approaches to semantic parsing. We will discuss how our formalisms handle them – or, in some cases, fail to handle them – in Chapters 5 and 6.

One phenomenon in language that is often regarded as having non-compositional aspects is **coreference**. Take the sentence *Harry doesn't want anyone to read his books*, and its AMR in Figure 3.13(a). The pronoun *his* can also refer to an expression outside the sentence, take for example *Voldemort is a dangerous writer. Harry doesn't want anyone to read his books.* Compare this to *Some people are protective*

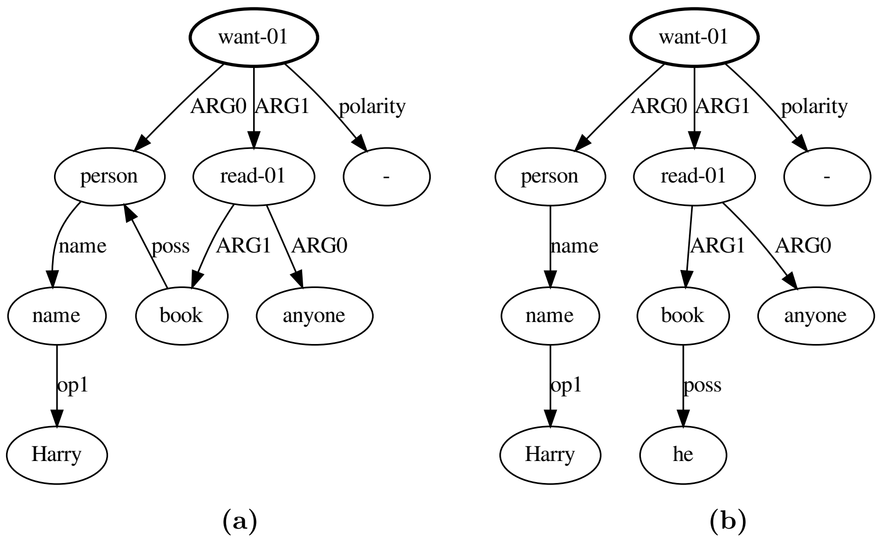


Figure 3.13: Two possible AMRs for *Harry doesn't want anyone to read his books.* In (a), the coreference is resolved sentence-internally, in (b) the pronoun *his* is interpreted as referring to someone outside the sentence.

about their belongings. Harry doesn't want anyone to read his books, and it becomes clear that who the pronoun *his* refers to depends on the context; the AMR in Figure 3.13(a) is only correct in the second context, in the first context the AMR in Figure 3.13(b) would be more appropriate. Generally, the context is not known in the AMR corpora. Still, the AMRs in the annotated corpora tend to resolve coreference whenever there is a viable antecedent in the sentence.

While binding theory describes syntactic constraints on who or what a pronoun can refer to, often that question cannot be answered by syntax alone. This poses a particular challenge to compositional models. In fact, as we will discuss in Chapter 5, this is a challenge we will not tackle in this thesis.

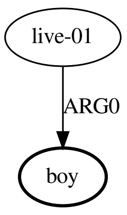
In a **relative clause**, a whole phrase acts as a modifier, using a slot that is usually used for argument application. For example in

(2) the boy who lived

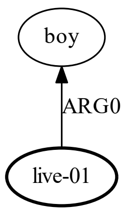
boy is the subject of *lived*, but this is achieved through modification rather than application. Figure 3.14 shows AMRs for (a) *the boy who lived* and (b) *the boy lived.* Note that the difference lies merely in the placement of the root (the bold outline).

Several phenomena can be understood as **movement**. For example in the question

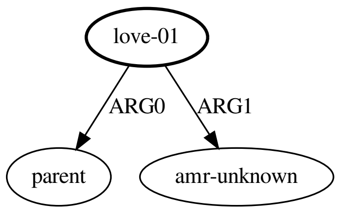
(3) a. Who$_i$ do the parents love $\__i$?



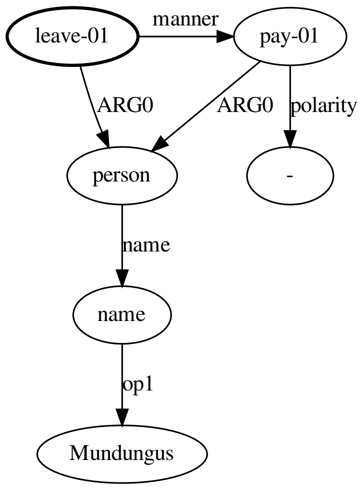
(a) The boy who lived



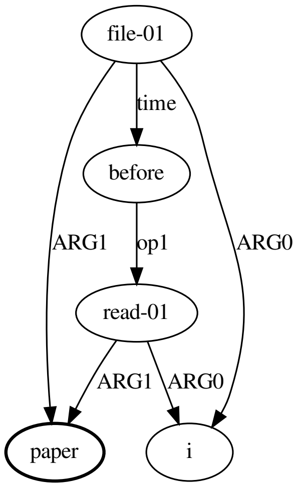
(b) The boy lived



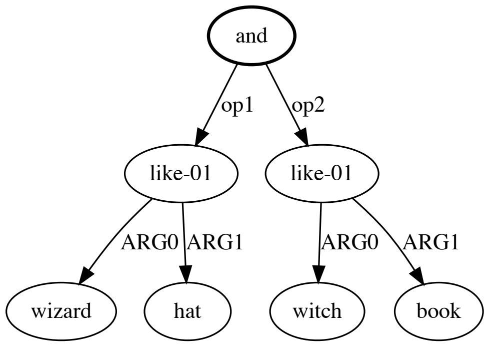
(c) Who do the parents love?



(d) Mundungus left without paying



(e) The paper I filed before reading



(f) Ellipsis: the wizard likes the hat and the witch the book

Figure 3.14: More example AMRs.

b. Who$_i$ does Malfoy doubt the parents love $\_\_i$?

c. Who$_i$ does Ron think Malfoy doubts the parents love $\_\_i$?

(AMR in Figure 3.14(c)), the meaning is straightforward (AMRs represent wh-words through *amr-unknown* labels). But while the word *who* takes the role of the object, it is not in the usual object position after the verb (marked with $\_\_i$). In fact, there is quite some distance between the word *who* and that object position. This distance is even greater in long-distance wh-movement as in (3-b) and (3-c). This presents a challenge to the syntax-semantics interface. This sort of displacement also occurs in e.g. raising, topicalization, and clefting.

Furthermore, control is not the only phenomenon with implicit arguments. The sentence

(4) Mundungus$_i$ left without $\_\_i$ paying

is an example of **secondary predication**, where *John* is the implicit subject of *paying* (see Figure 3.14(d)). In contrast to control, the common subject of the two verbs is a result of modification, not application. Similarly, a **parasitic gap** as in

(5) the paper$_k$ I$_i$ filed $\_\_k$ before $\_\_i$ reading $\_\_k$

yields both a common subject and object, see Figure 3.14(e).

Finally, consider the **ellipsis** in

(6) The wizard likes$_i$ hats and the witch $\_\_i$ books

and Figure 3.14(f). Here, the second occurrence of *likes* is omitted, indicated by '$\_\_i$'. But in the AMR, there are two *like-01* nodes, each with one ARG0 and one ARG1 edge, to properly specify who likes what. This example can be interpreted as both *like-01* nodes in the AMR being generated by the first occurrence of *likes*. In this interpretation the meaning of *likes* is duplicated, breaking the idea that each word contributes at most one part of the meaning.

3.3 Neural parsing

In the last few years, neural networks have taken the world of computational linguistics by storm. At their core, neural networks are very effective function approximators in high dimensional vector spaces, based on sequences of linear and nonlinear functions. I will give a technical discussion of neural networks in Chapter 6 when we use them in our parser. By encoding words or characters as vectors, neural networks can be employed in language tasks. A particularly impactful early application was Sutskever et al. (2014), who use a simple sequence-to-sequence (seq2seq) model: they have a recurrent neural network called LSTM simply read a sentence word by word, and produce a sentence in a different language again word by word. In between, there are only vector operations, and no use of linguistic structure.

Many neural models in NLP, such as the seq2seq models, don't make much explicit use of linguistic principles. Nonetheless, neural networks learn well from large amounts of data and perform language tasks surprisingly well. Seq2seq models have been employed for e.g. constituency parsing (Vinyals et al. (2015)) and semantic parsing (Dong and Lapata (2016)); seq2seq AMR parsers are discussed in Section 3.4 below. Some models combine neural networks with more structured decoding, such as Kiperwasser and Goldberg (2016) for dependency parsing, the tree decoder of Dong and Lapata (2016) for semantic parsing or Lewis and Steedman (2014) who use neural supertagging for CCG parsing.

Neural networks are an extremely powerful and flexible machine learning tool that yield simple yet effective models, with lower implementation costs than more traditional models such as synchronous grammars. However, they often require large amounts of training data (see the discussion of van Noord and Bos (2017) below). Further, their inner workings consisting of functions in high dimensional vector spaces are difficult to analyze and draw conclusions from.

3.4 Related work in AMR parsing

So far, we only discussed formalisms for semantic parsing. Now, let us look at some published AMR parsers (see Table 6.1 in Chapter 6 for evaluation scores). There are three groups of parsers that are particularly relevant for this thesis.

Synchronous grammars. First, there are the parsers based on synchronous grammars, such as Artzi et al. (2015) and Peng et al. (2015) (continued in Peng and Gildea (2016)). Artzi et al. (2015) pairs CCG rules on the syntax side with AMRs expressed as logical forms, such as

$$S \setminus NP : \lambda x.\lambda d.\text{dance-01}\,(d) \wedge \text{ARG0}\,(d, x)\,.$$

This example logical form means that there is a node d with label *dance-01*, and an edge with label ARG0 from d to the NP argument x. Peng et al. (2015) uses a synchronous HRG, which we previously discussed in Section 3.1.3. These models are inherently compositional, and strongly structured. While they, in particular

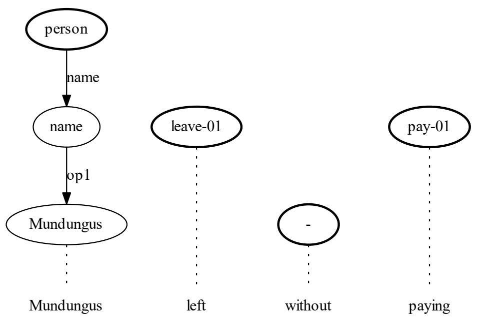
Artzi et al. (2015), performed strongly compared to other early AMR parsers, they have since been significantly outperformed by newer parsers, and no strong grammar based models have followed in their footsteps. As discussed in the introduction, the key challenge for these models is robustness: the rules they learn are very specific, and their strongly constrained derivation process is susceptible to irregularities and unseen or rare phenomena in the data.

Neural seq2seq models. The currently best performing neural seq2seq AMR parser, presented in van Noord and Bos (2017), is a simple yet effective neural sequence to sequence model. It reads the input sentence character by character, and outputs a string representation of the AMR character by character again. Thus, it makes virtually no assumptions on the relation between the sentence and the graph, being able to output effectively any string. Seq2seq models for AMR parsing struggle with data sparsity (Peng et al. (2017)). It is only through the use of 'silver' data, additional training data created by other parsers, that van Noord and Bos (2017) achieve competitive performance. Specifically, they pre-train their model on 100.000 sentence-graph pairs of silver data, in addition to the roughly 36.000 hand-annotated sentence-graph pairs in the largest AMR bank[8]. A central limitation of such an approach is that it relies on existing good parsers for the task that can generate the silver data – although, as van Noord and Bos (2017) show, it can then outperform those parsers.

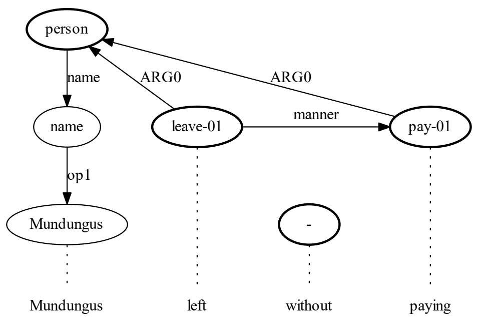
Graph decoders. The second group use what I will call here a *graph decoder*. This approach was pioneered by JAMR (Flanigan et al. (2014), Flanigan et al. (2016)) and further improved in Foland and Martin (2017) and Lyu and Titov (2018). These models have been proven very successful, with Flanigan et al. (2014) being the state of the art parser at the time when work on this thesis started, and Foland and Martin (2017) and Lyu and Titov (2018) setting new state of the art performance since through the addition of neural networks. As we will use this approach in a baseline model for the parser presented in Chapter 6, let us examine it in more detail here.

In a first *concept prediction* step, the graph decoder predicts a node or graph fragment for each word in the sentence, as in Figure 3.15(a) (this may be the empty graph for some words). These fragments represent the meaning of the word, but do not contain e.g. the ARGx edges of predicates. Each graph fragment has a marked root node (here again in bold) where the edges of the next steps attach, such as the *person* node for the *Mundungus* graph fragment.

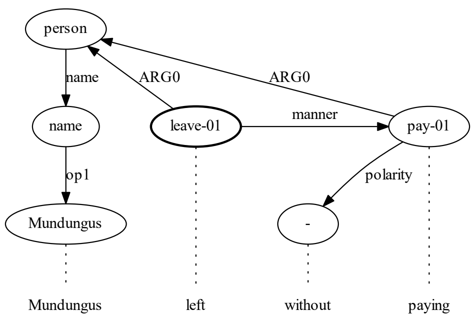
[8]LDC2017T10.



(a) Step 1: Concept prediction



(b) Step 2: Adding edges above threshold



(c) Step 3: Connecting the graph

Figure 3.15: A graph decoder predicts nodes and edges for *Mundungus left without paying.*

In a second step, all possible edges connecting the graph fragments are scored, and all edges above a fixed threshold are added. Typically, graph decoders allow only simple graphs, i.e. at most one edge between any two nodes. Further, every node is restricted to have at most one outgoing edge of each kind of predicate relation (ARG0, ARG1, . . .); see e.g. Flanigan et al. (2014) and Foland and Martin (2017). This step can create reentrancies; a possible result of this step is shown in Figure 3.15(b).

In a third step, repeatedly the best scoring edge that connects two components is added until the graph is connected. Inherently, this step cannot add reentrant edges. After this step, the graph is done and a root for the whole graph is chosen (Figure 3.15(c)).

The graph decoder models have more structure than the seq2seq models: they explicitly associate parts of the graph with the words in the sentence and can implement some restrictions on the graph structure as discussed above. Some other linguistically motivated ideas can be added; for example, Foland and Martin (2017) use two different neural networks to predict argument edges and non-argument edges. And indeed, graph decoders do not rely on additional silver data to perform well.

However, none of the published graph decoders for AMR have explicitly implemented the compositional ideas from earlier in this chapter, e.g. by modeling control or coordination with specialized mechanisms. While the models may well learn such mechanisms implicitly, they must learn them on their own and are not pushed towards them through a better inductive bias. And when such mechanisms are learned by a graph decoder, they remain encoded in the parser's learned parameters, i.e. in large vectors and matrices, which are difficult for humans to understand or interact with. We show in this thesis that adding explicit linguistic structure indeed helps with performance.

Others. CAMR (Wang et al. (2015), Wang et al. (2016)) transforms syntactic dependency trees into AMRs by e.g. deleting, relabeling and adding edges. Damonte et al. (2017), like the graph decoder models, predict graph fragments and edges between them, but uses a modified shift reduce parser to predict the edges.

3.5 Training a parser

In the last section, we discussed several approaches to AMR parsing. But how are these semantic parsers created in practice? Where do the grammar rules, the parameters to score them, the graph fragment lexicons, the feature weights, the parameters of the neural models come from? It is now widely believed in natural language processing that writing a complete broad-coverage model by hand is ineffective. Thus, the models are created through a combination of hand-written heuristics and *learning from a set of annotated training data*.

There are large AMR corpora available, building on each other, with the latest release (LDC2017T10) containing nearly 40,000 sentence-AMR pairs. These corpora contain sentences hand-annotated with their AMR (or more precisely, a linear representation of the AMR as discussed in Section 2.3). However, there is no annotation given as to how each AMR relates to its sentence, i.e. no compositional structure that would generate the AMR is given, nor alignments that denote which parts of the graph correspond to which word in the sentence.

While this lack of annotation is deliberate – the creators of the corpora did not want to force assumptions concerning the AMR parsing process onto other researchers – it poses a serious challenge to the learning of some models.

While this is no problem for the model of van Noord and Bos (2017) (it only needs the sentence-AMR pairs for training), the graph decoder models need to know which word creates which graph fragment in the training set. The models mostly rely on alignments obtained with statistical methods (Pourdamghani et al. (2014)) or heuristic rules (Flanigan et al. (2014)); Lyu and Titov (2018) learns alignments jointly with the parser parameters.

For compositional, grammar-based models, the problem is even more significant. Let us take a look again at the IRTG in Figure 3.1. To build a parser, one would need to have a large enough set of such rules to cover the large variety of sentences one may encounter in practical use, and a model to choose the best derivation tree for a sentence. A very simple version of such a model could be to associate each rule with a fixed score, and then, among the possible derivation trees for a sentence, choose the one with the highest sum of rule scores.

If the set of rules were given before training, and each sentence in the corpus annotated with a gold derivation tree, the rule weights could then be chosen to maximize the scores of the gold trees in the training data (while respecting some normalization constraint on the weights). Such maximum likelihood methods are standard for this task and well understood.

However, we would not want to write all rules for a broad-coverage grammar by hand, and prefer to learn them during training. Furthermore, the AMR corpora are not annotated with derivation trees. If we relax these conditions, and just assume that the training data were annotated with HR terms for the AMR, we could attempt to learn the rules automatically. The idea is to find repeating patterns in the terms and use these to create rules that generalize across many corpus entries. This process of finding rules is often done at the same time as optimizing their scores to fit the training data, so as to converge on a scored set of rules that explains the corpus well. A method for this is e.g. the one described in Cohn et al. (2009).

However, in reality, not even HR terms (or derivational terms of any kind) are given in the AMR corpora. The problem is that there are many possible terms for each AMR, and it is often unclear which one is correct and should be used for training. As we will see in Chapter 4, the number of possible terms in the HR algebra is in fact gigantic. Peng et al. (2015) face a similar issue when considering possible HRG terms. As a solution, they represent all such possible terms compactly in a structure much like a decomposition automaton. This allows them to sample terms from the compact representation, i.e. generate terms according to a probability distribution. This distribution starts out mostly uninformed, but whenever a term is generated, the probability distribution is adjusted to increase the likelihood of generating this or similar terms again. This makes terms in the future more likely to resemble previously generated terms, and the process is repeated several times over the corpus. This eventually provides terms that are consistent with each other, for all sentences in the training data.

Artzi et al. (2015) use a different, but also statistical method for inducing their lexical items, based on Kwiatkowski et al. (2010).

Another challenge to robustness. The heuristically created training data presents another challenge to the robustness of approaches based on synchronous grammars. They need to match syntactic structure to the terms that create the semantic representation, while respecting the alignments. Since none of syntactic structure, semantic terms or alignments are given in the corpora and thus all must be obtained heuristically, they don't always fit together. This complicates the training procedure, and e.g. Peng et al. (2015) note that their model is particularly sensitive to alignments. Artzi et al. (2015) employ an early update procedure if they cannot find a correct derivation for a training example, which occurs in nearly 40% of their training set. This robustness issue, paired with the fact that there are so many possible terms and only limited training data, mean that the problem of latent structural training data is far from solved.

3.6 This thesis' approach to semantic parsing

Let us recap the challenges put forth in the introduction.

Challenge 1: Models without linguistic structure are data hungry.

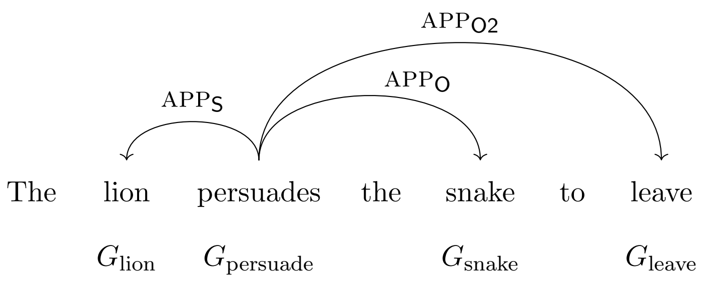
Challenge 2: It is not obvious how to add linguistic principles to neural networks.

Challenge 3: Synchronous grammars face robustness issues.

Challenge 4: Creating structural training data is a highly ambiguous process.

As we have seen, the principle of compositionality yields methods – in particular application, modification and unification – that seem well suited for semantic parsing. The recent successful neural approaches to semantic parsing do not use these principles, but could, a priori, benefit from them. Adding linguistic principles could make neural models more effective on the limited data we have (c.f. Challenge 1), but it is not clear how to do this effectively (Challenge 2). On the other hand, existing compositional parsers have problems with robustness and the lack of structural information in the training data – the Challenges 3 and 4.

We address these issues with the AM dependency approach in Chapter 6. This model uses dependency trees where each edge denotes an operation and words are associated with graph fragments, such as in



The operations are those of the AM algebra which we introduce in Chapter 5. The AM operations APP_α and MOD_α encode specific sequences of HR operations in one operation, to model application and modification as seen in Section 3.2, including unification mechanisms. Thus, the AM dependency parser uses methods based on compositionality. However, it drops the explicit syntax side of compositionality, and focuses purely on the semantic terms, using a flexible dependency approach. We still guide the machine learning model with established principles from linguistics, just more gently. As a result, we do not observe the same robustness issues a synchronous grammar would have, addressing Challenge 3.

We make use of neural supertagging methods (Lewis and Steedman (2014)) and neural dependency parsing methods (Kiperwasser and Goldberg (2016)) to predict the AM dependency trees. With these methods, we don't incorporate linguistic principles inside the neural networks, but use neural networks to predict linguistic structure. By not predicting the AMRs directly but using AM dependency trees as an in-between step, we give the neural models a target that is closer to the surface structure of the sentence, addressing Challenges 1 and 2. The AM algebra's type system further structures our approach. We show that predicting the AM dependency trees indeed improve performance, compared to a less structured graph decoder baseline that predicts the AMRs directly.

The thesis starts by addressing the training data issue, Challenge 4. Chapter 4 describes how to compute decomposition automata for the HR algebra, and examines whether they could be suitable for grammar induction via sampling, along the lines of Peng et al. (2015) as discussed above. While we find that there is so much ambiguity when describing an AMR with an HR term that the sampling approach is infeasible; the insights we gain lead to the AM algebra in Chapter 5. The AM algebra is based on the HR algebra and keeps many of its advantages, but while the HR algebra is a general tool for building graphs, the AM algebra is designed specifically for semantic graphs and draws on methods from compositional semantic parsing. In this manner, the AM algebra successfully generates much fewer candidate terms for a given graph, with only high quality terms remaining. Chapter 6 introduces the AM dependency parser. The AM dependency trees reduce the ambiguity when creating structured training data further, yielding only a single candidate dependency tree under certain conditions. Thus, the combination of the linguistically motivated AM algebra and the choice to predict semantic operations directly resolves Challenge 4 to the point where we don't even need to perform sampling any more.

The result is a parser that owes its strong performance to the effective combination of neural and compositional methods.

4

S-Graph Decomposition

In this chapter, we describe efficient algorithms to compute decomposition automata over the HR algebra (Courcelle and Engelfriet (2012), introduced in Section 2.4) for AMR. These decomposition automata (as described in Section 2.5.1) are a compact representation of all terms over the HR algebra that evaluate to a given AMR.

These decomposition automata have two key applications. First, we saw in Section 3.5 that to train a parser based on the HR algebra, we need terms for the AMRs in the training data; these terms are not given in the corpus, they are latent. The compact nature of decomposition automata allows their use in sampling techniques in the style of Peng et al. (2015), to obtain a consistent set of terms for the corpus. Second, HR decomposition automata can also be used in the IRTG parsing algorithm introduced in Section 3.1 to generate sentences from given AMRs. We will discuss this application and some related work briefly in Section 4.7. Further, since the HR algebra forms the backbone of the AM algebra we introduce in Chapter 5, the automata described in this section are also relevant for the dependency based model of Chapter 6, i.e. throughout the rest of this thesis.

The HR decomposition automata turn out to be complex objects, and we have to put much effort into computing them efficiently. We provide theoretical upper bounds and evaluate runtimes in practice. We also measure the size of their rule sets and languages. We find that there are so many ways to express an AMR as an HR term that even the compact decomposition automata become large and unwieldy objects, making e.g. grammar induction through sampling infeasible. We address that issue later in Chapter 5.

This chapter will provide the reader with (1) a thorough understanding of how the HR algebra can take AMRs apart and put them back together, (2) an efficient way of computing the HR decomposition automata, and (3) a quantification of remaining

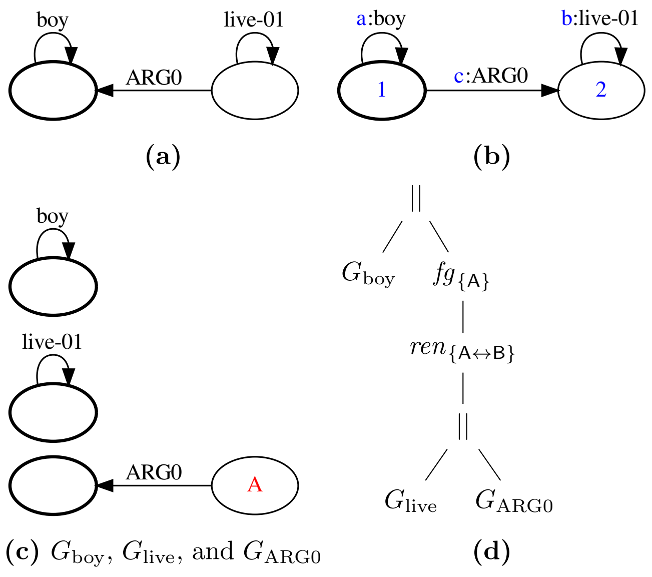


Figure 4.1: (a) An AMR $[G_0]_{iso}$ for the phrase *the boy who lived.*, (b) a concrete representative G_0, (c) atomic s-graphs and (d) an HR term that describes $[G_0]_{iso}$.

challenges to inducing a parsing model based on the HR algebra. We describe the problem setting in technical detail in Section 4.1 and examine the structure of the HR decomposition automata in Sections 4.2 and 4.3. We introduce our algorithms in Sections 4.4 and 4.5, and provide an empirical evaluation in Section 4.6.

Much of this chapter's content has been previously published in Groschwitz et al. (2015), and the paper and this chapter share some segments of text literally. It should also be mentioned that some content, mainly the idea of a boundary representation, has been inspired by the related work of Chiang et al. (2013). A closer comparison to Chiang et al. (2013) can be found in Section 4.7.

4.1 Problem setting

This section describes the task of this chapter in technical detail. Throughout this thesis, we work with AMRs that we model as abstract s-graphs, i.e. isomorphism classes such as $[G_0]_{iso}$ in Figure 4.1(a) of concrete graphs such as G_0 in Figure 4.1(b). Recall from Section 2.2 that concrete graphs specify node and edge identities (blue in the figures), while abstract graphs do not. This technical distinction is crucial in the HR algebra (recall Section 2.4), and also in this chapter.

Let now $[G]_{iso}$ be an AMR (or generally, a connected abstract s-graph). Throughout this section we fix G as a concrete representative. The goal of this chapter is to describe, and efficiently compute, the decomposition automaton for $[G]_{iso}$ with respect to the HR algebra (Courcelle and Engelfriet (2012), for short C&E).

Recall from Section 2.2 that we encode node labels as loops, that is, all information of the graph is encoded in its edges. We therefore choose an HR signature where the constants are single loops and single edges, such as the ones in Figure 4.1(c). This mirrors the approach of C&E. Further, recall that AMRs are rooted, and that we represent the root with a source R. Throughout this section, we then work with a finite set of source names $\mathcal{S}$, such that $\mathsf{R} \in \mathcal{S}$. As constants, we use the set

$$\begin{aligned}\mathfrak{C}_{[G]_{iso}} = &\left\{\mathbf{a}^{\ell}_{\lambda} \mid a \in \mathcal{S}, \lambda \text{ a label of a loop in } [G]_{iso}\right\}\\ &\cup \left\{\mathbf{ab}_{\lambda} \mid a, b \in \mathcal{S}, \lambda \text{ a label of a non-loop edge in } [G]_{iso}\right\},\end{aligned}$$

where $\mathbf{a}^{\ell}_{\lambda}$ and $\mathbf{ab}_{\lambda}$ are the single loops and single edges, with sources on their incident nodes, as described in Definition 2.16. Throughout this chapter we will use the signature $\Sigma_{\mathcal{S},\mathfrak{C}_{[G]_{iso}}}$ as defined in Definition 2.15, with constants in $\mathfrak{C}_G$ and all other HR operations (forget, rename and merge) with respect to the source set $\mathcal{S}$. We write the shorthand Σ for $\Sigma_{\mathcal{S},\mathfrak{C}_{[G]_{iso}}}$. Finally note that all AMRs are connected, and consequently we assume the graph $[G]_{iso}$ we decompose to be connected.

Let HR_Σ be the HR algebra with signature Σ. An example term of HR_Σ that evaluates to the graph $[G_0]_{iso}$ of Figure 4.1(a) is shown in Figure 4.1(d). Our task in this section then is to find a decomposition automaton D of $[G]_{iso}$ with respect to HR_Σ, i.e. an automaton D whose language consists of all terms that evaluate to $[G]_{iso}$, that is

$$L(D) = \left\{t \in T(\Sigma) \mid [\![t]\!]_{\mathrm{HR}_\Sigma} = [G]_{iso}\right\}.$$

4.2 HR decomposition automata.

The question then is, given a connected abstract s-graph $[G]_{iso}$, what does such an HR decomposition automaton look like? We note a core idea of the string decomposition automaton of Section 2.5.1, where the states were spans, i.e. smaller parts of the full sentence. The analogue here is to use subgraphs as states, to keep track of which parts of the final graph $[G]_{iso}$ a term has covered. To facilitate our reasoning about subgraphs, we work on an (arbitrary) concrete representative G of our abstract graph $[G]_{iso}$. Specifically, we use *sub-s-graphs* as states (recall from Section 2.4 that sub-s-graphs are subgraphs, with potentially different source assignments). Without further ado, let us consider the following definition as a candidate for a decomposition automaton.

$$\begin{aligned}
G_{\text{boy}} &\rightarrow (\{1\}, \{a\}, \{1 \mapsto \mathsf{R}\})\\
G_{\text{live}} &\rightarrow (\{2\}, \{b\}, \{2 \mapsto \mathsf{R}\})\\
G_{\text{ARG0}} &\rightarrow (\{1,2\}, \{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})\\
|| ((\{2\}, \{b\}, \{2 \mapsto \mathsf{R}\}), (\{1,2\}, \{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})) &\rightarrow (\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})\\
ren_{\{\mathsf{A} \leftrightarrow \mathsf{R}\}} ((\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})) &\rightarrow (\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{R}, 2 \mapsto \mathsf{A}\})\\
fg_{\{\mathsf{A}\}} ((\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{R}, 2 \mapsto \mathsf{A}\})) &\rightarrow (\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{R}\})\\
|| ((\{1\}, \{a\}, \{1 \mapsto \mathsf{R}\}), (\{1,2\}, \{b,c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})) &\rightarrow (\{1,2\}, \{a,b,c\}, \{1 \mapsto \mathsf{R}\})
\end{aligned}$$

Figure 4.2: Example rules of the decomposition automaton D for the AMR $[G_0]_{iso}$, using the concrete representative in Figure 4.1(b) and the constants in Figure 4.1(c). These rules can be used to build the term in Figure 4.1(d). The states, i.e. sub-s-graphs, are represented as triples of node set, edge set and source assignment.

Definition 4.1. Let G be a connected concrete s-graph, $\mathcal{S}$a finite set of sources. Let Σ be an HR signature over $\mathcal{S}$, here specifically the signature as defined in Section 4.1. Then we define the automaton $D = \langle Q, \Sigma, Q_f, \Delta \rangle$ where the set of states Q is the set of sub-s-graphs of G, and the set of final states $Q_f = \{G\}$ contains only G. Δ is the set of the following transition rules:

- $|| \, (H_1, H_2) \to H_3$ for all $H_1, H_2, H_3 \in Q$ with $H_3 = H_1 \, || \, H_2$.
- $ren_h \, (H_1) \to H_2$ for every permutation h of $\mathcal{S}$ and all $H_1, H_2 \in Q$ with $H_2 = ren_h \, (H_1)$.
- $fg_B \, (H_1) \to H_2$ for every subset $B \subseteq \mathcal{S}$ and all $H_1, H_2 \in Q$ with $H_2 = fg_B \, (H_1)$.
- $c \to H$ for every $c \in \Sigma$ of arity 0, and every $H \in Q$ with $H \simeq c$.

In other words, this automaton describes how *concrete* sub-s-graphs can combine to the full graph G. Example rules of the decomposition automaton of the AMR in Figure 4.1(a), that allow building the term in Figure 4.1(c), are shown in Figure 4.2. For example, the rule

$$G_{\text{live}} \to (\{2\}, \{b\}, \{2 \mapsto \mathsf{R}\})$$

for G_{live} evaluates to the corresponding concrete subgraph of G_0,

$$\{2\}, \{b\}, \{2 \mapsto \mathsf{R}\}$$

which is a triple of node set, edge set and source assignment function. The rule

$$G_{\text{ARG0}} \to (\{1, 2\}, \{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})$$

works similarly. We can combine these graphs with the rule

$$\begin{aligned} || \, ((\{2\}, \{b\}, \{2 \mapsto \mathsf{R}\}), (\{1, 2\}, \{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})) \\ \to (\{1, 2\}, \{b, c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\}). \end{aligned}$$

This means the automaton assigns the state

$$(\{1, 2\}, \{b, c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\}),$$

consisting of the live-01 and ARG0 edges with source A at node 1 and R at 2, to the term $G_{\text{live}} \, || \, G_{\text{ARG0}}$. This correctly identifies a subgraph of G_0 that matches the term.

We now show that the automaton we just defined indeed defines the correct language. First, we show this crucial proposition that relates the concrete subgraph states of D to abstract graphs derived by the HR algebra.

Proposition 4.2. *Let H be a sub-s-graph of G, D as defined in Definition 4.1 and $t \in T(\Sigma)$, i.e. t is a term over the signature of D. Then*

$$t \xrightarrow[D]{*} H \Leftrightarrow [\![t]\!]_{HR} = [H]_{iso}$$

Proof. We show that

$$t \xrightarrow[D]{*} H \Leftrightarrow [\![t]\!]_{\mathrm{HR}} = [H]_{iso}$$

by induction over the height of t.

We first show that '$\Rightarrow$' holds. So let us assume that $t \xrightarrow[D]{*} H$, i.e. D can assign H to the top of t. Let first t have height 1, i.e. $t = c$ for some constant c, then $H \simeq c$ by definition of D, and $[\![t]\!]_{\mathrm{HR}} = [\![c]\!]_{\mathrm{HR}} = [H]_{iso}$ holds.

Let us now assume that the statement holds for height $n-1$, and let t have height $n > 1$. Consider the case $t = \,||\,(t_1, t_2)$ for some terms t_1, t_2. Since $t \xrightarrow[D]{*} H$, there are sub-s-graphs H_1, H_2 derived by these subterms, i.e. $t_1 \xrightarrow[D]{*} H_1$, $t_2 \xrightarrow[D]{*} H_2$, such that $H = H_1 \,||\, H_2$ (the operation interpreted for concrete s-graphs). By induction then, $[\![t_i]\!]_{\mathrm{HR}} = [H_i]_{iso}$ for $i = 1, 2$. Therefore then

$$\begin{aligned}
[\![t]\!]_{\mathrm{HR}} &= [\![\,||\,(t_1, t_2)]\!]_{\mathrm{HR}} \\
&= [\![t_1]\!]_{\mathrm{HR}} \;||\; [\![t_2]\!]_{\mathrm{HR}} \\
&= [H_1]_{iso} \;||\; [H_2]_{iso} \\
&= [H]_{iso} \, ,
\end{aligned}$$

where the operations are evaluated on abstract s-graphs, and the last step follows from $H = H_1 \,||\, H_2$. A similar argument can be made for the cases $t = \mathit{ren}_h(t_1)$ and $t = \mathit{fg}_B(t_1)$ for any rename and forget operations.

We now show that '$\Leftarrow$' holds. We assume $[\![t]\!]_{\mathrm{HR}} = [H]_{iso}$, i.e. the term evaluates to $[H]_{iso}$ in the HR algebra. Again, if t has height 1, i.e. $t = c$ for some constant c, we get $t \xrightarrow[D]{*} H$ immediately from $H \simeq c$.

Again, we assume then that the statement holds for height $n-1$, let t have height $n > 1$ and consider the case $t = \,||\,(t_1, t_2)$ for some terms t_1, t_2. Then $[\![t]\!]_{\mathrm{HR}} = [H]_{iso}$ plus the definition of $||$ for abstract graphs gives us the existence of two concrete s-graphs H_1, H_2 (not necessarily sub-s-graphs of G) such that $[\![t_i]\!]\,\mathrm{HR} = [H_i]_{iso}$ for $i = 1, 2$, and $[\![t]\!]_{\mathrm{HR}} = [H_1 \,||\, H_2]_{iso}$. In particular $H \simeq H_1 \,||\, H_2$, i.e. there is an isomorphism f such that $f(H_1 \,||\, H_2) = H$. Then $H_i \simeq f(H_i) \subseteq H$, $i = 1, 2$ (note that the restriction of f to H_i is still injective, and obviously surjective to $f(H_i)$). Firstly then, we have $[\![t_i]\!]_{\mathrm{HR}} = [H_i]_{iso}$ for $i = 1, 2$. Secondly, $f(H_i) \subseteq H$, $i = 1, 2$ implies that the $f(H_i)$ are sub-s-graphs of G, and by induction we obtain

$$t_i \xrightarrow[D]{*} f(H_i), i = 1, 2. \tag{4.1}$$

Finally, it is easy to see that $H = f(H_1 \mid\mid H_2) = f(H_1) \mid\mid f(H_2)$. Combined with (4.1), this yields $t \xrightarrow[D]{*} H$, which we set out to prove. Again, a similar argument works if t was built with rename or forget.

□

The following corollary states that Definition 4.1 is correct. It follows immediately from Proposition 4.2 for the case $H = G$.

Corollary 4.3. *The automaton in Definition 4.1 is indeed a decomposition automaton for $[G]_{iso}$, i.e. for any $[G]_{iso}$, G and D as in Definition 4.1,*

$$L(D) = \{t \in T(\Sigma) \mid [\![t]\!]_{HR} = [G]_{iso}\}$$

4.2.1 Computing the decomposition automaton

In order to work with the HR decomposition automata, we need to compute them first – unfortunately they don't emerge from the graph G directly. This problem spurs all work in the rest of this chapter.

Recall from Section 2.5.2 that in order to compute the full automaton, we only need to be able to answer either of the following type of queries:

- *Top-down:* given a sub-s-graph H and an algebra operation f, enumerate all the rules $H \to f(H, \ldots, H_k)$ in D. This asks how a larger sub-s-graph can be derived from other sub-s-graphs using the operation f.
- *Bottom-up:* given sub-s-graphs $H_1, \ldots, H_k$ and an algebra operation f, enumerate all the rules $H \to f(H_1, \ldots, H_k)$ in D. This asks how smaller sub-s-graphs can be combined into a bigger one using the operation f. Note that here, if the operation is not a constant, it is deterministic and, there can be at most one such rule for the query (constants can evaluate to multiple different isomorphic sub-s-graphs).

In Sections 4.4 and 4.5, we will describe how to answer these queries, and use the efficient bottom-up and top-down algorithms of Section 2.5.2 to compute the full automaton. We evaluate the algorithms in practice in Section 4.6.

But first, we need to better understand the inherent structure of sub-s-graphs, the central objects of our compositional analysis, in the next Section 4.3.

4.3 Structure of sub-s-graphs

In this section, we will find that only a certain class of sub-s-graphs needs to be considered as states in our decomposition automata. This observation leads to more compact representations of the states, which will prove crucial in both practical implementation and asymptotic runtime analysis.

Recall from Section 2.5 that for an automaton A over signature Σ with states Q, we call a state $q \in Q$ *accessible* if there is a ground term $t \in T(\Sigma)$ such that $t \xrightarrow[A]{*} q$, i.e. there is a term that derives the state q. We call a state $q \in Q$ *extensible* if there is a term $t \in T(\Sigma \cup Q)$ and a final state $q_f \in Q_f$ with $t \xrightarrow[A]{*} q_f$, such that q is a leaf of t. That is, if we start with q, we can use operations, and at leaves other states, to reach a final state. For a state to participate in a successful run, i.e. for it to be relevant to the automaton's language, it must be both accessible and extensible.

Here, since we only need our automata to produce the right language, we focus on reachable and extensible states. These classes of states turn out to have useful properties.

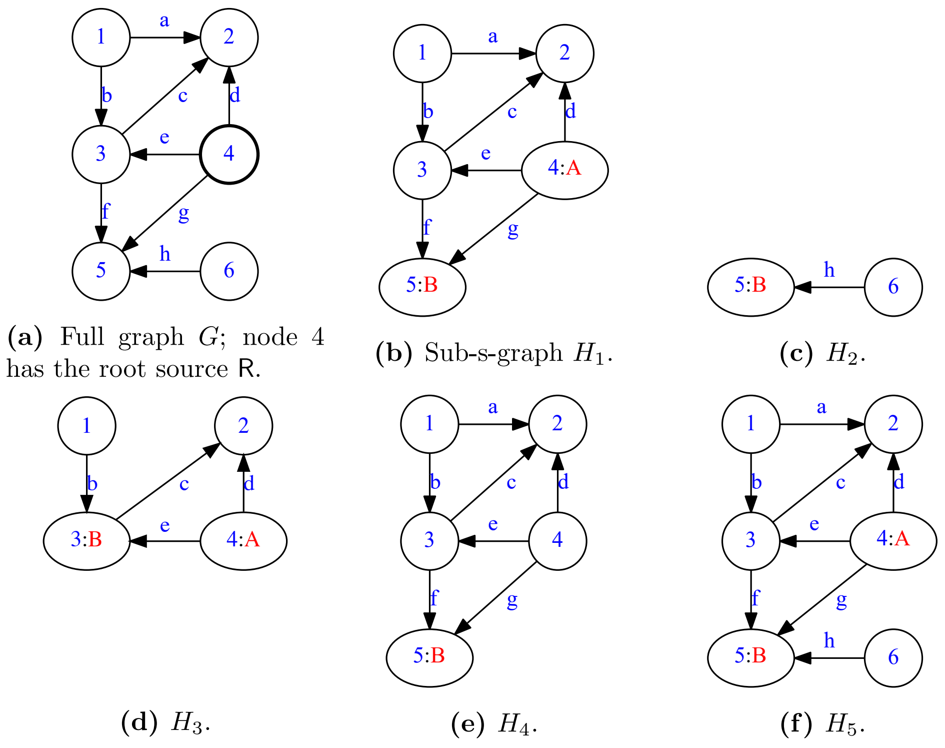
First note that all reachable states of D are graphs that are induced by a set of edges (i.e., there are no isolated nodes). This is because all the constants in the signature have that property (single loops and edges, paired with their incident nodes), and none of the HR operations can change that. We can therefore focus mostly on just edges and source assignments, an insight that permeates this section throughout.

We will first look at the *s-components* of a graph G, a new idea of this work. Let $U \subseteq V_G$ be a set of source nodes, i.e. $U = D(\phi)$, for some source assignment ϕ. This set splits the edges of G into equivalence classes that are separated by U. We say that two edges $e, f \in E_G$ are equivalent with respect to U, $e \sim_U f$, if there is a sequence

$$v_1 \overset{e}{\leftrightarrow} v_2 \leftrightarrow \dots v_{k-1} \overset{f}{\leftrightarrow} v_k$$

with $v_2, \dots, v_{k-1} \notin U$, i.e. if we can reach f from an endpoint of e without visiting a node in U. We call the equivalence classes of E_G with respect to $\sim_U$ the *s-components* of G and denote the s- component that contains an edge e with $[e]_{\sim_U}$.

Consider for example the graph G in Figure 4.3, with sources A and B as in H_5 in Figure 4.3(f). Here, the set of source nodes is $U = \{4, 5\}$. Starting at edge g, it is impossible to reach another edge without crossing one of those nodes; the same



(a) Full graph G; node 4 has the root source R.

(b) Sub-s-graph H_1.

(c) H_2.

(d) H_3.

(e) H_4.

(f) H_5.

Figure 4.3: (a) An s-graph G with (b-f) some sub-s-graphs.

holds for h. The other edges are however connected through non-source nodes, for example one can get from d to f with the sequence

$$4 \overset{d}{\leftrightarrow} 2 \overset{c}{\leftrightarrow} 3 \overset{f}{\leftrightarrow} 5$$

(the outer nodes of the path may be source nodes). Thus, the s-components of H_5 in Figure 4.3(f) are $\{a, b, c, d, e, f\}$, $\{g\}$ and $\{h\}$.

The notion of s-components is of particular importance due to the following lemma.

Lemma 4.4. *Let H a sub-s-graph of G that is a reachable state in D. Then H is an extensible state in D if and only if its edge set is the union of a set of s-components of G with respect to the source assignment ϕ_H of H, and all source nodes in G that are nodes of H are also source nodes in H, i.e. $Src\,(G) \cap V_H \subseteq Src\,(H)$.*

For example, the sub-s-graph H_1 in Figure 4.3(b) is extensible. It fulfills the conditions, and indeed, if we merge it with H_2 (in (c) in the figure), forget the B source and rename A to R, we obtain the full graph G in (a). In contrast, the sub-s-graph H_3 shown in Figure 4.3(d) is not extensible. The s-components of G with the source assignment of H_3 are $\{a, b, c, d\}$, $\{e\}$ and $\{f, g, h\}$. In H_3, the edge a from the s-component $\{a, b, c, d\}$ is missing, and there is no way to add it without sources at 1 and 2 (and no way to add sources there). The impossibility to add sources to a node is also why H_4 in Figure 4.3(e) is not extensible. We can merge it with H_2 to have all edges and nodes of G, but G has an R source at 4, and this can't be added to H_4 with the HR algebra. A formal proof based on these ideas is presented in the Appendix A.

Since every reachable state is a subgraph induced by its edge set, we can represent every reachable state via its edges and sources alone. We let an *s-component representation* $\mathcal{C} = (C, \phi_{\mathcal{C}})$ consist of a source assignment $\phi_{\mathcal{C}}$ and a non-empty set C of s-components of G with respect to the set $D(\phi_{\mathcal{C}})$ of source nodes of $\phi_{\mathcal{C}}$. We require the domain $D(\phi_{\mathcal{C}})$ to be contained in the subgraph induced by the edges in C. Then we can represent every extensible sub-s-graph $H = (H^\circ, \phi_H)$ of G by the s-component representation $\mathcal{C} = (C, \phi_H)$ where C is the set of s-components of which H consists. Conversely, we write $T(\mathcal{C})$ for the unique reachable and extensible sub-s-graph of G represented by the s-component representation $\mathcal{C}$. I.e. for $\mathcal{C} = (C, \phi_{\mathcal{C}})$, the graph of $T(\mathcal{C})$ is the one induced by the edges in C, and its sources are defined by $\phi_{\mathcal{C}}$. The sub-s-graph H_1 in Figure 4.3(b) has the s-component representation $(\{\{a, b, c, d, e, f\}, \{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})$.

In summary, every reachable and extensible state of $\mathrm{A}_{G,\mathcal{S}}$ can be uniquely represented by an s-component representation. Since only such states can participate in a successful run of an automaton, we only consider these for the rest of this chapter. These more compact representations allow us to see structural properties of sub-s-graphs more clearly.

The main utility of s-component representations derives from the fact that merge can be evaluated on these representations alone, as follows.

Lemma 4.5. *Let $\mathcal{C} = (C, \phi_{\mathcal{C}})$, $\mathcal{C}_1 = (C_1, \phi_{\mathcal{C}_1})$, $\mathcal{C}_2 = (C_2, \phi_{\mathcal{C}_2})$ be s-component representations in the s-graph G. Then $T(\mathcal{C}) = T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ if and only if*

(i) $\phi_{\mathcal{C}} = \phi_{\mathcal{C}_1} \cup \phi_{\mathcal{C}_2}$ (in particular, $\phi_{\mathcal{C}_1}$ and $\phi_{\mathcal{C}_2}$ must be compatible).

(ii) All sets in C_1 and C_2 are s-components with respect to $D(\phi_{\mathcal{C}})$.

(iii) $C = C_1 \uplus C_2$ (i.e., disjoint union).

We write $\mathcal{C} = \mathcal{C}_1 \mid\mid \mathcal{C}_2$. If there is no such $\mathcal{C}$, then $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ is not defined.

Note that Condition (ii) is in fact implied by (iii) and $\mathcal{C}$ being an s-component representation, but it appears important enough to mention it explicitly.

For example, take again the sub-s-graph H_1 of Figure 4.3(b), with s-component representation

$$(\{\{a, b, c, d, e, f\}, \{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})$$

as seen above. We can merge it with H_2 from Figure 4.3(c), that we represent as

$$(\{\{h\}\}, \{5 \mapsto \mathsf{B}\}) .$$

Now observe that choosing the boundary representation

$$(\{\{a, b, c, d, e, f\}, \{g\}, \{h\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})$$

as the result $\mathcal{C}$ satisfies all conditions, and indeed represents the result of the merge, H_5 in Figure 4.3(f).

Trying to merge H_1 with K_1 of Figure 4.4 fails. We cannot merge these concrete graphs because the sources at node 5 don't match. And indeed, when considering the s-component representation

$$(\{\{h\}\}, \{5 \mapsto \mathsf{C}\})$$

of K_1, we can see that Condition (i) of the lemma fails here, since ϕ_{H_1} and ϕ_{K_1} are not compatible.

Merging H_1 with K_2 of Figure 4.4 doesn't work either, simply because the graphs overlap (both in edges, and in non-source nodes of H_1). Condition (i) holds here: the source assignments are compatible, and their union is

$$\{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}, 2 \mapsto \mathsf{C}, 3 \mapsto \mathsf{D}\} .$$

However, Condition (ii) fails. The s-component $\{a, b, c, d, e, f\}$ of H_1 is not an s-component with respect to that new source assignment.

Finally, trying to merge H_1 with K_3 fails again due to overlap. In the lemma, Conditions (i) and (ii) hold, but Condition (iii) fails: K_3 is represented as

$$(\{\{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})$$

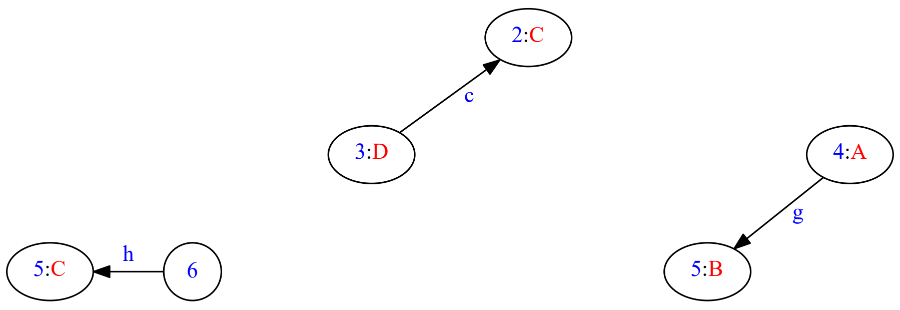
and now both the representations for H_1 and K_3 contain the component $\{g\}$; the union $C = C_1 \cup C_2$ of the component sets is not disjoint.

In the Appendix A, we present a proof based on the ideas in the examples.

4.3.1 Boundary representations

In order to check the conditions of Lemma 4.5 efficiently in the bottom-up algorithm of Section 4.4, it will be useful to represent s-components via their *boundary*.

Consider an s-component representation $\mathcal{C} = (C, \phi_{\mathcal{C}})$ in G and let $E_{\mathcal{C}}$ be the



(a) K_1. **(b)** K_2. **(c)** K_3.

Figure 4.4: More sub-s-graphs of G from Figure 4.3, that all cannot be merged with H_1.

set of all edges that are adjacent to a source node in $D(\phi_{\mathcal{C}})$ and contained in an s-component in C. Then we let the *boundary representation (boundary representation)* β of $\mathcal{C}$ in the s-graph G be the pair $\beta = (E_{\mathcal{C}}, \phi_{\mathcal{C}})$. That is, β represents the s-components through the *in-boundary edges*, i.e. those edges inside the s-components (and thus the sub-s-graph) which are adjacent to a source. As an example, the boundary representation of H_1 is

$$\beta_{H_1} = (\{d, e, f, g\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\}).$$

The boundary representation β specifies $\mathcal{C}$ uniquely and we can compute $\mathcal{C}$ from β, as the following lemma shows. Thus we can write $T(\beta)$ for $T(\mathcal{C})$.

Lemma 4.6. *Let $\mathcal{C} = (C, \phi_{\mathcal{C}})$ be an s-component representation, and $\beta = \beta(\mathcal{C}) = (E_\beta, \phi_{\mathcal{C}})$ be its boundary representation. Then if $\phi_{\mathcal{C}}$ is empty, C is just one s-component spanning the whole graph G, and otherwise*

$$\mathcal{C} = \left(\left\{[e]_{\sim_{D(\phi_{\mathcal{C}})}} \mid e \in E_\beta\right\}, \phi_{\mathcal{C}}\right).$$

In other words, we can simply determine which s-components to use based on the edges in β.

Proof. The case where $\phi_{\mathcal{C}}$ is empty is trivial. In the other case, let us denote the set $\left\{[e]_{\sim_{D(\phi_{\mathcal{C}})}} \mid e \in E_\beta\right\}$ with C_β. It suffices to show that $C = C_\beta$. That '$\supseteq$'holds is obvious from the construction of boundary representations. To see that '$\subseteq$' holds, note that since G is connected and $\phi_{\mathcal{C}}$ non-empty, every s-component M in Cmust touch a source node. Each such source node in turn must have an adjacent edge e in one of the s-components in C, which implies that e is in the boundary representation,

$e \in E_\beta$. But then $M = [e]_{\sim_{D(\phi_C)}} \in C_\beta$. □

This means that we can freely translate between boundary representations, s-component representations and (reachable and extensible) sub-s-graphs, without losing information.

The following lemma shows that the merge operation can be evaluated directly on the boundary representations.

Lemma 4.7. *Let G be an s-graph, let $\beta = (E, \phi), \beta_1 = (E_1, \phi_{\beta_1}), \beta_2 = (E_2, \phi_{\beta_2})$ be boundary representations in G. Then $T(\beta) = T(\beta_1) \mid\mid T(\beta_2)$ if and only if the following conditions hold:*

(i) $\phi = \phi_{\beta_1} \cup \phi_{\beta_2}$;

(ii) for every source node v of β_1 that is not a source node in β_2, the last edge on the shortest path in G from v to the closest source node of β_2 is not an in-boundary edge of β_2, and vice versa;

(iii) $E = E_1 \uplus E_2$, i.e. the disjoint union.

We write $\beta = \beta_1 \mid\mid \beta_2$. If no such β exists, $T(\beta_1) \mid\mid T(\beta_2)$ is undefined.

Intuitively, the conditions (ii) and (iii) guarantee that the component sets are disjoint; the lemma then follows from Lemma 4.5 (with a formal proof in the Appendix A). We can again illustrate the conditions with the example graphs H_1 (Figure 4.3(b)) and K_1, K_2, K_3 (Figure 4.4). Merging H_1 and K_1 fails as before, violating again Condition (i) for the source assignments.

The case of trying to merge H_1 and K_2 is more interesting. As seen above, we have

$$\beta_{H_1} = (\{d, e, f, g\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\}),$$

and the boundary representation of K_2 is

$$\beta_{K_2} = (\{c\}, \{2 \mapsto \mathsf{C}, 3 \mapsto \mathsf{D}\}).$$

Here, Condition (ii) is violated. For example, the node 2 is a source node in β_{K_2} but not in β_{H_1}. The last edge on the shortest path from 2 to a source node of β_{H_1}, i.e. to 4 or 5, is the edge d to node 4. This edge d however is an in-boundary edge of β_{H_1}, and the condition fails.

The boundary representation for K_3,

$$\beta_{K_3} = (\{g\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})$$

$$\frac{}{\beta} \quad \langle \text{There is a constant } c \in \Sigma \text{ with } T(\beta) \simeq [\![c]\!] \rangle$$

$$\frac{\beta}{ren_h(\beta)} \quad \langle h \text{ is a permutation of } \mathcal{S} \rangle$$

$$\frac{\beta}{fg_B(\beta)} \quad \langle B \text{ is a subset of } \mathcal{S} \rangle$$

$$\frac{\beta_1, \beta_2}{\beta_1 \mid\mid \beta_2} \quad \langle \beta_1 \mid\mid \beta_2 \text{ is defined as by Lemma 4.7} \rangle$$

Figure 4.5: Deduction schema for bottom-up exploration

clearly violates Condition (iii) when combined with β_{H_1}, since the sets of in-boundary edges $E_{\beta_{K_3}} = \{g\}$ and $E_{\beta_{H_1}} = \{d, e, f, g\}$ overlap.

We can also evaluate the other HR operations directly on boundary representations. **Constants** are straightforward, and so are the **rename** operations, which only change the source assignment ϕ.

The **forget** operation changes the domain of ϕ, so some in-boundary edges may no longer be on the boundary afterwards. However, note that we can restrict ourselves, to forget sources only on nodes where all incident edges are in the graph. Otherwise, the result would not be extensible by Lemma 4.4. And in these cases, we can simply first remove the sources in question from ϕ, and then remove all in-boundary edges that are no longer on the boundary.

We can thus interpret the operations of the HR algebra as (partial) functions on boundary representations, leaving them undefined if the result would be non-extensible and thus irrelevant for the decomposition automaton. Note how the boundary representations are simpler than full graphs or even s-component representations, dropping the interior edges that we can easily reconstruct. This simplification will allow us to make the bottom-up algorithm in the next section efficient.

4.4 Bottom-up algorithm

With the work done in the previous section, we can now tackle computing the decomposition automaton D efficiently. We use the algorithm of Section 2.5.2, when we originally discussed tree automata, to obtain the explicit automaton from bottom-up queries. Recall the deduction schema of Figure 2.10, which is spelled out for the HR algebra in Figure 4.5. We sequentially derive boundary representations, first using constants as axioms to derive matching boundary representations in the graph, and then repeatedly applying HR operations to already derived boundary representations. Whenever we make a deduction, we add the corresponding rule to the explicit automaton we build.

This approach comes with two challenges:

1. for each operation, test efficiently whether the operation is allowed and if applicable which state, i.e. boundary representation it produces, and

2. for the merge operation, efficiently look up which previously seen states a new state can combine with.

We use the standard Algorithm 2 of Section 2.5.2 to run through the schema of Figure 4.5, here spelled out as Algorithm 3[1]. To recap, the general idea is to add all states found by constants (the first inference rule in Figure 4.5) to an agenda and a lookup structure. We then keep pulling states from the agenda, applying the rename, forget and merge operations, and add new states to the agenda and the lookup structure. For testing the merge operation, we find suitable previously derived merge siblings in the lookup structure.

We show that this bottom-up algorithm has a total runtime of $O\left(n^s 3^{ds} ds\right)$, where n is the number of nodes in G, d its maximal degree and s the number of sources in the HR algebra we use. We describe details on how we execute each HR operation in Section 4.4.2. But first, we describe the lookup structure for the merge operation.

4.4.1 Lookup

The lookup step for potential merge partners is crucial for the algorithm's runtime. First, observe that if we have a total of $s = |\mathcal{S}|$ sources, then there are $O\left(n^s 2^{ds}\right)$ boundary representations in G, where d is the maximum degree in G. This is because there are $O\left(n^s\right)$ possible source assignments, and if there are s source nodes, for each of them, each of the adjacent up to d edges can be in the in-boundary set or not (depending on the structure of the graph). Now, if we would test *every* pair of boundary representations for whether they can be merged, we would get $O\left(\left(n^s 2^{ds}\right)^2\right)$ such tests. However, there are much fewer actually possible merge operations $\beta = \beta_1 \mid\mid \beta_2$. We can see this by focusing on the resulting boundary operation β. It too has a source assignment with up to s sources, of which there are $O\left(n^s\right)$ many. Then every edge adjacent to a source node could have been in either β_1, or β_2, or neither. This gives $O\left(3^{ds}\right)$ options. And in fact, the knowledge of which boundary edges were in β_1 and β_2, paired with the source assignment of β,

[1]With a slight simplification, since lookup is only necessary for the merge operation, and that operation is symmetrical.

Algorithm 3 Bottom-up exploration

```
seen := $\emptyset$, agenda := (), $\Delta := \emptyset$
lookup := new lookup structure for merge
for $c \in \Sigma, \mathsf{ar}(c) = 0$ do
    for $\beta$ boundary representation in $G$, $T(\beta) \simeq [\![c]\!]$ do
        add $c \to T(\beta)$ to $\Delta$
        if $\beta \notin$ seen then
            add $\beta$ to seen
            add $\beta$ to agenda
            add $\beta$ to lookup
        end if
    end for
end for
while agenda $\neq \emptyset$ do
    pop $\beta$ from agenda
    for $f \in \Sigma, \mathsf{ar}(f) = 1$ do
        if $f(\beta)$ defined then
            $\beta' := f(\beta)$
            add $f(T(\beta)) \to T(\beta')$ to $\Delta$
            if $\beta' \notin$ seen then
                add $\beta'$ to seen
                add $\beta'$ to agenda
                add $\beta'$ to lookup
            end if
        end if
    end for
    for $\beta' \in$ lookup$(\beta)$ do
        if $\beta \mid\mid \beta'$ defined then
            $\beta'' := \beta \mid\mid \beta'$
            add $\mid\mid(T(\beta)\,T(\beta')) \to T(\beta'')$ to $\Delta$
            add $\mid\mid(T(\beta')\,T(\beta)) \to T(\beta'')$ to $\Delta$
            if $\beta'' \notin$ seen then
                add $\beta''$ to seen
                add $\beta''$ to agenda
                add $\beta''$ to lookup
            end if
        end if
    end for
end while
return $\Delta$
```

fully defines β_1 and β_2 and thus the operation. Therefore, there are only $O\left(n^s 3^{ds}\right)$ merge operations, much fewer than the $O\left(\left(n^s 2^{ds}\right)^2\right)$ total pairings β_1, β_2.

We now describe a lookup structure that allows us to perform only $O\left(n^s 3^{ds}\right)$

tests. The general structure of lookup is a tree as shown in Figure 4.6. At each leaf, there is another simple lookup structure (e.g. a hash table) based on sets of in-boundary edges. When a boundary representation β is entered into lookup, we traverse the tree top-down until we find the right node to sort β into. When we do a lookup (Line 26), we collect possible partners from all relevant nodes.

Sorting. When we enter a boundary representation $\beta = (E_\beta, \phi_\beta)$ into lookup, we start at the top of the tree t_V in Figure 4.6. We then go through the sources $\alpha_1, \ldots, \alpha_s$ in $\mathcal{S}$ in order.[2] If $\phi_\beta^{-1}(\alpha_1) = v_j$, i.e. if ϕ_β assigns to v_j the source α_1, then we choose the branch v_j. If ϕ_β leaves α_1 unassigned, we choose the branch $\perp$. We continue this way for all $\alpha_i, i = 1, \ldots, s$. Once we reach a leaf of t_V in this fashion, the path to this leaf fully encodes the source assignment ϕ_β. Then, at the leaf, we have a straightforward indexing structure (as said above, e.g. a hash table) that maps sets of in-boundary edges to boundary representations. We simply enter $E_\beta \mapsto \beta$ here.

Figure 4.7(a) shows the lookup tree t_V for the graph G_0 of Figure 4.1(a) with sources $\mathcal{S} = \{\mathsf{R}, \mathsf{A}\}$. The yellow path shows how the boundary representation $(\{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})$ obtained from evaluating the constant G_{ARG0} is sorted in; the blue path shows the process for the boundary representation $(\{b\}, \{2 \mapsto \mathsf{R}\})$ obtained from G_{live}

Partner lookup. While sorting was deterministic, when looking up potential partners for merge, we need to be careful to find *all* of them. We do this the following way. Given a boundary representation $\beta = (E_\beta, \phi_\beta)$, to find all possible partners $\beta' \in \mathsf{lookup}(\beta)$, we first traverse the tree t_V of Figure 4.6 to collect multiple leafs. At each juncture, we now follow multiple paths. If ϕ_β assigns α_i to a node v_j, we follow the paths labeled v_j and $\perp$. This mirrors the fact that we can merge β with a boundary representation that has α_i also assigned to v_j or unassigned, but not with one that has α_i assigned to a different node. If however ϕ_β leaves α_i unassigned, we follow *all* branches downward, since all of them may contain a boundary representation with which a merge is possible. Once we reach a leaf, note that this path defines the set M of source nodes of a possible merge partner β' (namely, exactly the set of nodes we encountered along the path), and thus also the set of edges E_M incident to a vertex in M. None of the edges of β may be in β', so in the table at this leaf, we look up the boundary representations for each subset $F \subseteq E_M \setminus E$. We return all boundary representations found that way as possible partners β'.

[2]We can choose an arbitrary order at the start of decomposition

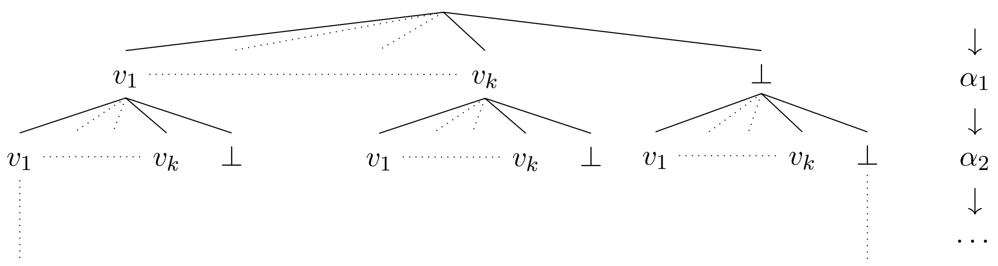
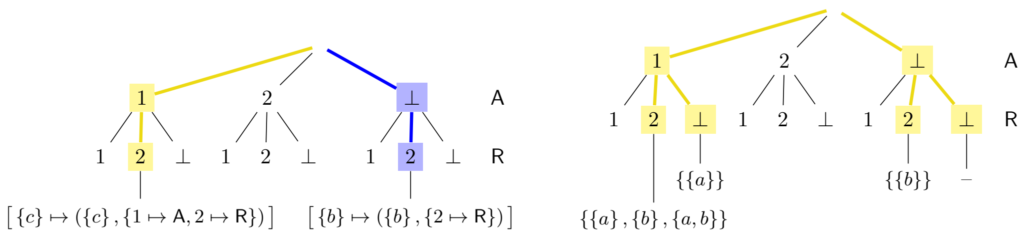


Figure 4.6: Node lookup tree t_V



A
R
$[\{c\} \mapsto (\{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})]$
$[\{b\} \mapsto (\{b\}, \{2 \mapsto \mathsf{R}\})]$
$\{\{a\}, \{b\}, \{a, b\}\}$
$\{\{a\}\}$
$\{\{b\}\}$
–

(a) Sorting **(b)** Lookup

Figure 4.7: Example lookup procedure when decomposing G_0 of Figure 4.1(a) with sources $\mathcal{S} = \{\mathsf{R}, \mathsf{A}\}$.

Figure 4.7(b) shows all paths that the lookup for the case $\beta = (\{c\}, \{1 \mapsto \mathsf{A}, 2 \mapsto \mathsf{R}\})$ (again, the boundary representation obtained from evaluating the constant G_{ARG0}) follows, as well as the sets F we use for lookup at every leaf. We have $E_\beta = \{c\}$, and for example at path $(\mathsf{A}/1, \mathsf{R}/2)$ – indicating that source A is at node 1 and R is at node 2, we have $M = \{1, 2\}$ and thus $E_M \{a, b, c\}$ and $E_M \setminus E = \{a, b\}$, leaving us with options $\{a\}$, $\{b\}$ and $\{a, b\}$ for F. Along the path $(\mathsf{A}/\perp, \mathsf{R}/2)$ – indicating that A is unassigned, and R is at 2, we can find the boundary representation $(\{b\}, \{2 \mapsto \mathsf{R}\})$ obtained from G_{live}; c.f. the blue path in Figure 4.7(a).

Note that each such pairing β, β' is characterized by a consistent source function ϕ (the union of ϕ_β and the path we follow during lookup to obtain β'), and two sets of edges that are disjoint and contain only edges incident to source nodes of ϕ. By the same reasoning we used to count merge operations, we obtain that in total there can be at most $O\left(n^s 3^{ds}\right)$ such pairs.

Note that without the edge-set lookup tables at the leaves, the number of pairs would be higher. The source assignments of any pair β, β' would still be consistent, but for each edge incident to a source node, there are now 4 options: in β, in β', in neither, or *in both*. That is, there would be $O\left(n^s 4^{ds}\right)$ pairs.

We have thus defined an efficient lookup structure, and can move on to analyze the runtime of the full algorithm.

	forget	rename	merge
I	$O\left(n^s 2^{ds} 2^s\right)$	$O\left(n^s 2^{ds} s!\right)$	$O\left(n^s 3^{ds}\right)$
T	$O\left(ds\right)$	$O\left(s\right)$	$O\left(ds\right)$

Table 4.1: Rule counts I and amortized per-rule runtimes T for the different rule types in the bottom-up algorithm.

4.4.2 Answering queries and runtime analysis

The runtime of Algorithm 3 is bounded by the number I of different operations (or rules) we need to test (Lines 4, 16 and 27), multiplied by the *per-rule runtime* T that we need to answer each query. The factor I is analogous to the number of rule instances in schema-based parsing . The factor T is often ignored in the analysis of such schema, because e.g. in parsing schemata for strings, we typically have $T = O(1)$. This is not the case here, however. Table 4.1 summarizes our results. The merge operations dominate the runtime, bringing the total asymptotic runtime to $O\left(n^s 3^{ds} ds\right)$. A closer discussion of each operation follows.

Forget and rename. Given a boundary representation $\beta = (E, \phi_\beta)$, answering the bottom-up forget query $fg_B\left(\beta'\right) \to \beta$ for a set of sources B amounts to verifying that for every $\alpha \in B$, all edges incident to $\phi_\beta(\alpha)$ are in-boundary in β, since otherwise the result would not be extensible as discussed earlier. This takes time $O\left(d\right)$. We then let $\beta' = (E', \phi_{\beta'})$, where $\phi_{\beta'}$ is like $\phi_{\beta'}$ but undefined on $\phi_\beta^{-1}\left(B\right)$, and E' is the set of edges in E that are still incident to a source in $\phi_{\beta'}$. Computing β' thus takes time $O\left(d\right)$.

For a rename operation ren_h, we do not need to perform any checks, and simply concatenate ϕ_β and h, yielding a per-rule runtime $O\left(s\right)$.

We pull each of the $O\left(\left(n2^d\right)^s\right)$ different boundary representations from the agenda once, and apply each of the $O\left(s!\right)$ rename and $O\left(2^s\right)$ forget operations once to it. Since the result of a forget or rename rule is determined by the child β', this is an upper bound for the number I of rule instances of forget or rename.

Merge. Now consider the bottom-up merge query for the boundary representations β_1 and β_2. As we saw in Section 4.3.1, $T\left(\beta_1\right) \;||\; T\left(\beta_2\right)$ is not always defined. But if it is, we can answer the query with the rule $(\beta_1 \;||\; \beta_2) \to \;||\; (\beta_1, \beta_2)$, with $\beta_1 \;||\; \beta_2$ defined as in Section 4.3.1. Computing this boundary representation takes time $O\left(ds\right)$.

We can check whether $T\left(\beta_1\right) \;||\; T\left(\beta_2\right)$ is defined by going through the conditions of Lemma 4.7. The only nontrivial condition is (ii). In order to check it efficiently, we precompute a data structure which contains, for any two nodes $u, v \in V$, the length k of the shortest undirected path $u = v_1 \leftrightarrow \ldots \overset{e}{\leftrightarrow} v_k = v$ and the last edge e on this path. This can be done in time $O\left(n^3\right)$ using the Floyd-Warshall algorithm (Floyd (1962)). Checking (ii) for each source then takes time $O(s)$ per rule.

Given that there are a total of $O\left(n^s 3^{ds}\right)$ merge rules, the runtime for checking (ii) amortizes to $O(s)$ per rule. The Floyd-Warshall step amortizes to $O(1)$ per rule for $s \geq 3$. Note that for $s = 2$, the HR algebra can only generate trees, and for $s = 1$ only strings of edges. If G falls in these categories, the node table can be easily computed in amortized $O(1)$ using trivial algorithms[3]. This yields a total amortized per-rule runtime T for bottom-up merge of $O(ds)$.

We can also include the runtime for the lookup structure here. Note that the tree t_V has depth s, and a hash table lookup / addition is of cost $O(1)$ on average. Thus, we get a runtime of $O\left(\left(n2^d\right)^s s\right)$ for sorting, because we sort each graph once, and each step has cost $O(s)$. For the lookup, we saw in the previous section that it returns, in total, $O\left(n^s 3^{ds}\right)$ pairings. Note that the reasoning of last section actually applies to all lookups we *try*, not only the ones where actually a graph is found[4]. Each lookup costs again at most $O(s)$, which is subsumed in the per-rule runtime $O(ds)$ for merge we established above.

Constants. Since all the constants in Σ consist of single edges or loops, we can find all pairs β, c such that $T(\beta) \simeq [\![c]\!]$ in one pass over the graph, taking time linear in the size of G, i.e. $O(nd)$.

We have now shown the operation specific asymptotic runtimes of Table 4.1, and the total runtime of $O\left(n^s 3^{ds} ds\right)$ follows for the bottom-up case. We now turn to the top-down case.

4.5 Top-down algorithm

As seen back in Section 2.5.2, we can also explore the rules of the automaton the other way around, in top-down direction. This circumvents the lookup problem, since instead of having to find *all* possible partners in the merge operation, we now know the result of the merge and need to 'divide it up' instead. As we will show, this algorithm has the same asymptotic runtime as the bottom-up case; the

[3]If s is this small, but G is not a tree / string of edges, then the language $L(D)$ of the decomposition automaton is empty, and we can just leave the rule set Δempty as well.

[4]Note that we perform the lookup twice for every pair of boundary representations β_1, β_2, once when we pull β_1 from the agenda, and another time when we pull β_2; only the one we perform last will be successful. However, this only adds a constant factor to the runtime and does not change the asymptotic analysis.

$$\frac{\mathcal{C}}{} \quad \langle \text{There is a constant } c \in \Sigma \text{ with } T(\mathcal{C}) \simeq [\![c]\!] \rangle$$

$$\frac{\mathcal{C}}{\mathcal{C}'} \quad \langle \mathcal{C}' = ren_h(\mathcal{C}) \text{ for a permutation } h \text{ of } \mathcal{S} \rangle$$

$$\frac{\mathcal{C}}{\mathcal{C}'} \quad \langle \mathcal{C}' = fg_B(\mathcal{C}) \text{ for a subset } B \text{ of } \mathcal{S} \rangle$$

$$\frac{\mathcal{C}}{\mathcal{C}_1, \mathcal{C}_2} \quad \langle \mathcal{C} = \mathcal{C}_1 \mid\mid \mathcal{C}_2 \text{ as by Lemma 4.5} \rangle$$

Figure 4.8: Deduction schema for top-down exploration. Note that the first rule for constants doesn't actually derive any new state, but we can still check it and infer the corresponding rule.

complexities broken down by operation are again as in Table 4.1 and the total runtime is $O\left(n^s 3^{ds} ds\right)$.

We follow the standard Algorithm 1 for the deduction schema in Figure 2.9; the schema for this case is spelled out in Figure 4.8. The idea is to start with the final state (the full graph) and repeatedly discover new child states through top-down queries. Again, whenever we make a deduction, we add the corresponding rule to the explicit automaton we build. We do not use boundary representations here, but s-component representations.

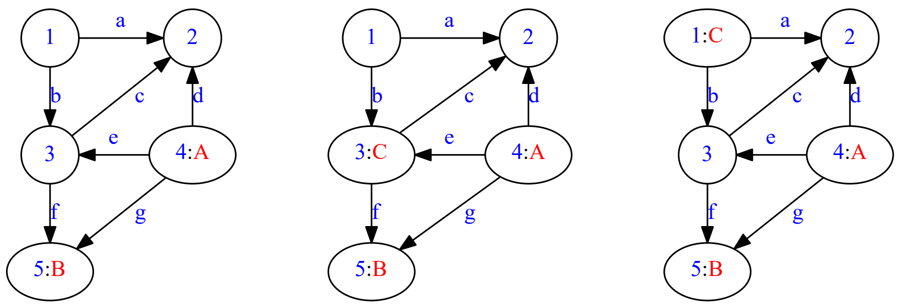
Crucial for the runtime here is to answer the queries efficiently. We describe the queries in the upcoming section, and analyze the asymptotic runtime after that.

4.5.1 Answering queries

Constants. Since the constants are single edges, this check is trivial.

Merge. Given an s-component representation $\mathcal{C} = (C, \phi_\mathcal{C})$, by Lemma 4.5, we can enumerate all s-component representations $\mathcal{C}_1$ and $\mathcal{C}_2$ that can be merged to yield $\mathcal{C}$, by using every distribution of the s-components in C over $\mathcal{C}_1$ and $\mathcal{C}_2$ and restricting ϕ accordingly.

Rename. Given an s-component representation $\mathcal{C} = (C, \phi_\mathcal{C})$ and a rename operation ren_h, finding the s-component $\mathcal{C}'$ such that $ren_h(\mathcal{C}') = \mathcal{C}$ is in fact the deterministic process of simply concatenating $\phi_\mathcal{C}$ with the inverse h^{-1} of the permutation h.



(a) H_1. **(b)** K. **(c)** L.

Figure 4.9: Examples to illustrate the 'inverse' forget procedure. We have $H_1 = fg_{\{\mathsf{C}\}}(K) = fg_{\{\mathsf{C}\}}(L)$

Forget. The challenging query to answer top-down is forget. We will first describe the problem and introduce a data structure that supports efficient top-down forget queries.

Consider top-down forget queries on the sub-s-graph H_1 in Figure 4.3(b) (reprinted in Figure 4.9 for convenience); its s-component representation is

$$(\{\{a, b, c, d, e, f\}, \{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}\})\,.$$

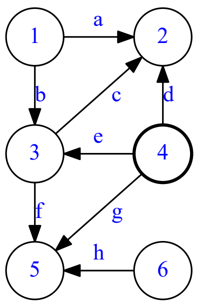
A top-down forget might promote the node 3 to a C-source, yielding a sub-s-graph K (that is, $fg_{\{\mathsf{C}\}}(K)$ is the original s-graph H_1; K is also shown in Figure 4.9). In K, the edges a, e, and f are no longer equivalent; its s-component representation is

$$(\{\{a, b, c, d\}, \{e\}, \{f\}, \{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}, 3 \mapsto \mathsf{C}\})\,.$$

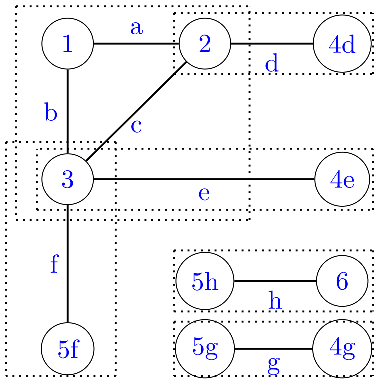
Thus, promoting 3 to a source splits the original s-component into smaller parts.

By contrast, the same top-down forget might instead promote the node 1 to a C-source, yielding the sub-s-graph L in Figure 4.9; $fg_{\{\mathsf{C}\}}(L)$ is also H_1. However, all edges in $\{a, b, c, d, e, f\}$ are still equivalent in L; its s-component representation is $(\{\{a, b, c, d, e, f\}, \{g\}\}, \{4 \mapsto \mathsf{A}, 5 \mapsto \mathsf{B}, 1 \mapsto \mathsf{C}\})$, having the same s-components as H_1.

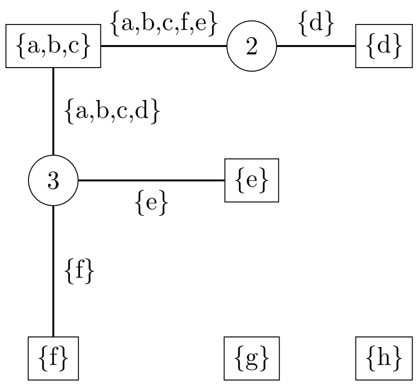
An algorithm for top-down forget must be able to determine whether promotion of a node splits an s-component or not. To do this, let G be the input graph. We create an undirected auxiliary graph G^U from G and a set U of (source) nodes. G^U contains all nodes in $V \backslash U$, and for each edge e that is incident to a node $u \in U$, it contains a node (u, e). Furthermore, G^U contains undirected versions of all edges in G; if an edge $e \in E$ is incident to a node $u \in U$, it becomes incident to (u, e) in G^U instead. The auxiliary graph $G^{\{4,5\}}$ for our example graph is shown in Fig. 4.10(b).



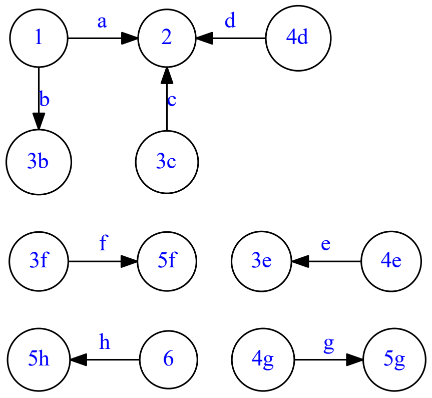
(a) G



(b) $G^{\{4,5\}}$



(c) Block-cutpoint graph of $G^{\{4,5\}}$



(d) $G^{\{4,5,6\}}$

Figure 4.10: (a) Our running example graph G, (b) the auxiliary graph $G^{\{4,5\}}$, (c) the block-cutpoint graph of $G^{\{4,5\}}$, and (d) the auxiliary graph $G^{\{3,4,5\}}$.

Two edges are connected in G^U if and only if they are equivalent with respect to U in G – by 'splitting' the source nodes, paths actually *cannot* go through them any more. In other words, the connected components of G^U are the s-components of G with respect to U. Therefore, promotion of u splits s-components iff u is a *cutpoint* in G^U, i.e. a node whose removal disconnects its connected component[5]. A *biconnected component (BCC)* of G^U is a maximal subgraph H such that any node can be removed from H without disconnecting it. Two overlapping BCCs always share just a single node, and that node is a cutpoint – otherwise, the BCCs would not be maximal, they could be merged without losing their defining property. In Fig. 4.10(b), the BCCs are indicated by the dotted boxes. Observe that 3 is a cutpoint and 1 is not.

For any given U, we can represent the structure of the BCCs of G^U in its *block-cutpoint graph*. This is a bipartite graph whose nodes are the cutpoints and BCCs of G^U, and a cutpoint is connected to all the BCCs that contain it; see Fig. 4.10(c) for the block-cutpoint graph of G^U in the example (ignoring its edge labels for now).

Connectivity in a block-cutpoint graph mirrors connectivity in the original graph (here G^U), and thus a block-cutpoint graph is always a forest: if there were a cycle, everything in it could be contracted into a single BCC. The individual trees thus represent the connected components of G^U, i.e. the s-components of G.

This line of thought leads to a useful property of block-cutpoint graphs. At a node that represents a cutpoint u, every incident edge leads to a subtree of the block-cutpoint graph representing a new connected component that would emerge if we were to remove u (or 'split' it). For example, if we were to promote the node 3 to a source, it would be split into $3b, 3c, 3e, 3f$ in $G^{\{3,4,5\}}$ (see Figure 4.10(d)). Note that the edges b and c are still connected, and part of the component $\{a, b, c, d\}$. This component corresponds to the bi-connected components $\{a, b, c\}$ and $\{d\}$, represented in the subtree of the block-cutpoint graph in Figure 4.10(c) that one obtains when following the edge from node 3 upwards, i.e. the subtree consisting of the nodes labeled $\{a, b, c\}$, 2 and $\{d\}$. We annotate this edge from node 3 upwards with all edges we find in that direction, i.e. with $\{a, b, c, d\}$. We do the same for the other edges, always annotating an edge from a cutpoint to a BCC with the set of all edges we encounter in the subtree attached to the BCC (in the direction away from the cutpoint). For example, for the other edges incident to 3, this yields the sets $\{e\}$ and $\{f\}$, so the edges incident to 3 are annotated with exactly the new s-components we obtain when promoting 3 to a source.

We precompute the annotated block-cutpoint graphs for all relevant sets U; de-

[5]Note that removing a node u disconnects a component of G^U if and only if repeating our 'splitting' process for u disconnects the component.

tails are presented in the next section.

We can now answer a top-down forget query $\mathcal{C} \to fg_B\mathcal{C}'$ efficiently from the block-cutpoint graph for the sources of $\mathcal{C} = (C, \phi)$. We start with some $\alpha \in B$. We iterate over all components $c \in C$, and then over all internal nodes u of c. If u is not a cutpoint, we simply let $\mathcal{C}' = (C', \phi_{\mathcal{C}'})$ by making u an α-source in $\phi_{\mathcal{C}'}$ and letting $C' = C$. Otherwise, we also remove c from C and add the new s-components on the edges adjacent to u in the block-cutpoint graph. We repeat the process for all sources in B. The query returns rules for all s-component representations that can be constructed like this.

4.5.2 Runtime analysis

Merge and rename. Since merge is so straightforwardly generated by distributing the s-components of the parent on to the children, the cost lies essentially only in the generation of the s-component representation data structures. This brings the per-rule runtime to $O(ds)$, the maximum number of s-components in C. The number of merge rules is, by the same reasoning as in the bottom-up case, again $O\left(n^s 3^{ds}\right)$. For rename, the same analysis as in the bottom-up case applies.

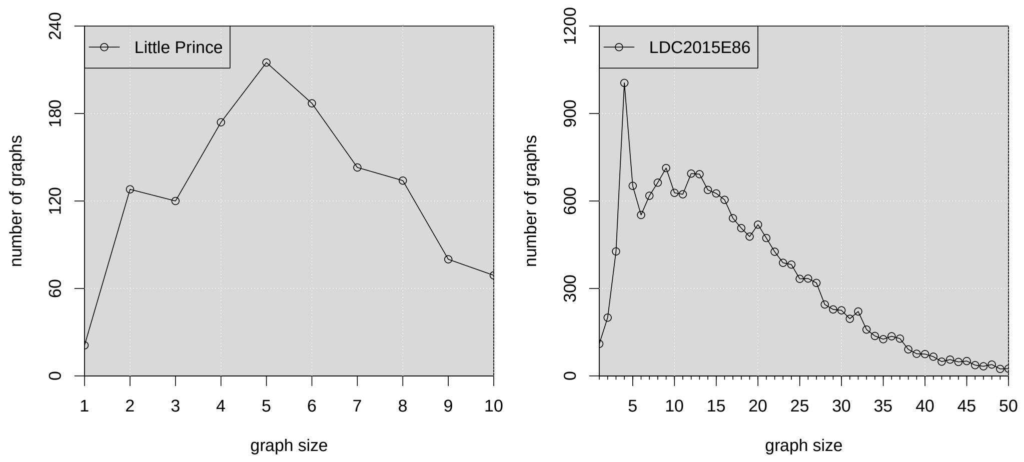
Forget. Once the block-cutpoint graphs are precomputed, the runtime of top-down forget fg_B is $O\left(ds\right)$, since the set B is of size $O\left(s\right)$ and for each source, we may add $O\left(d\right)$ new s-components. We only need to precompute the block-cutpoint graphs with respect to all sets $U \subseteq V$ of nodes with $|U| \leq s-1$, since only in these cases can another source be added. There are $O\left(n^{s-1}\right)$ such graphs. Each step of the computation is at most a linear run through the graph, see e.g. Hopcroft and Tarjan (1973) for computing the biconnected components. These linear runs can be upper-bound as $O\left(nd\right)$. The pre-computation thus takes $O\left(n^s d\right)$ time, which amortizes to $O\left(1\right)$ per rule.

Constants. Again, this check is trivial to perform.

In conclusion, we have shown that the asymptotic runtimes of Table 4.1 also hold in the top-down case, and the total asymptotic runtime of $O\left(n^s 3^{ds} ds\right)$ follows.

4.6 Evaluation

In this section, we evaluate the two algorithms introduced in this chapter in practice. We examine runtimes, automata sizes, language sizes and coverage. Runtimes are



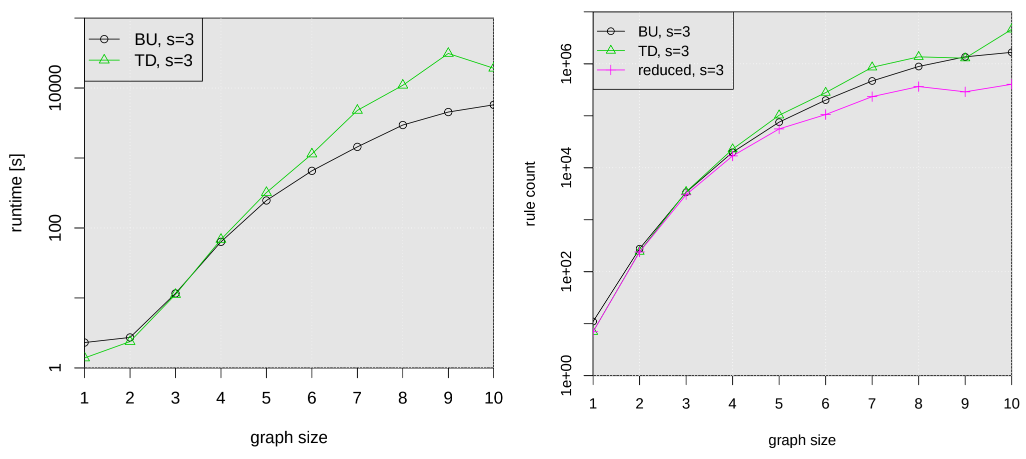
(a) Little Prince corpus **(b)** LDC2015E86 corpus

Figure 4.11: AMR size distributions on the corpora we use.

measured for enumerating all rules in the automata with the given algorithms. We use fixed finite source sets that contain the root source R, and arbitrary other sources (we just use numbers here). We use a slight variation of the HR algebra here, which simplified implementation. First, rename operations can only swap two source names instead of performing arbitrary permutations. Any permutation can still be performed through a sequence of swaps. Second, forget operations only forget one source at a time. Again, sets of sources can be forgotten through sequences of rename operations.

First, we experiment with different versions and parameters on the Little Prince corpus (version 1.6), which is available at `https://amr.isi.edu/download/amr-bank-v1.6.txt`. This is a small corpus of 1,562 sentences; we perform our experiments on the 1,271 sentences where the AMR has up to 10 nodes. The distribution of AMR sizes is shown in Figure 4.11(a). At the end of this section, we will test the best versions on the training section of the LDC2015E86 dataset, which has 16,833 sentences; we use the 16,616 sentences where the AMR has up to 50 nodes (see Figure 4.11 for the size distribution).

Bottom-up versus top-down. Figure 4.12 shows in (a) a runtime comparison of the bottom-up and top-down algorithms, each using the HR algebra with 3 sources and constants that are single edges or single labeled nodes. For these plots, we use geometric means across all graphs with the same node count, and plot on log scales. We can see that the top-down algorithm is a bit slower. Figure 4.12 shows a possible reason. It shows all rules that the algorithms explore, and for comparison the number of rules that participate in a successful run, i.e. the minimal number of rules



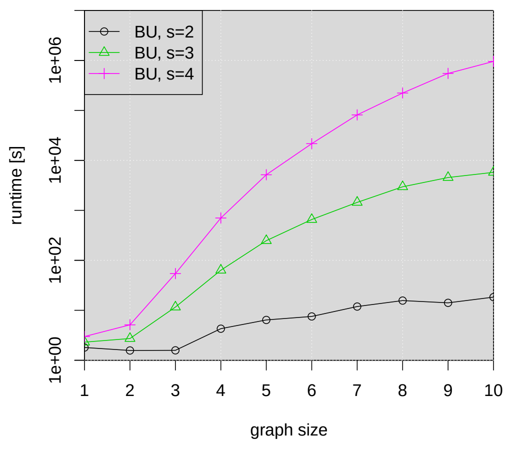
(a) Runtimes

(b) Automata size; *reduced* shows the number of rules that participate in successful runs of the automaton.

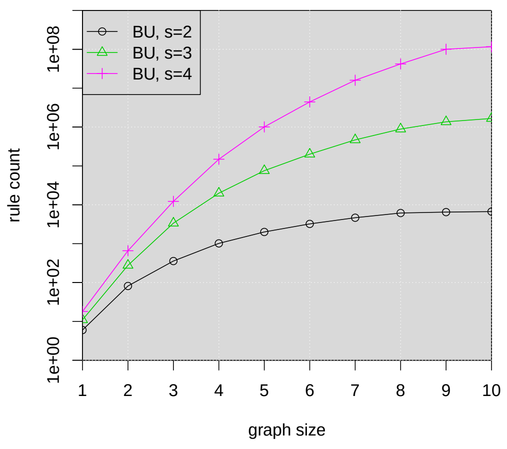
Figure 4.12: Comparing the bottom-up and top-down runtimes, with 3 sources.

necessary to compute the correct language. It seems like the top-down algorithm explores a few more rules that in the end cannot be used. The top-down algorithm may still be useful for downstream applications of the decomposition automaton that require the automaton to answer top-down queries lazily. That is for example the case in some IRTG parsing algorithms. However, in this thesis we care mostly about enumerating all rules. Thus, we will use bottom-up automata in the further evaluation.

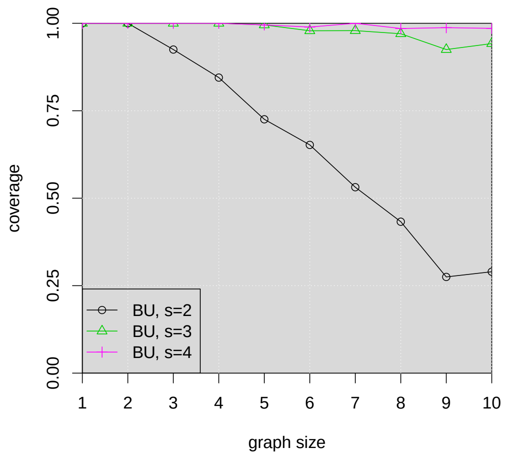
Source count. We evaluate the bottom-up algorithm with 2,3 and 4 sources, see Figure 4.13. The asymptotic runtime $O\left(n^s 3^{ds} ds\right)$ has the source count s in the exponent, and it is thus no surprise that increasing the number of sources comes at a great cost in terms of runtimes. As Figure 4.13(a) shows, just adding one source increases the average time to decompose an AMR with 10 nodes by a factor of over 100. For four sources, this means several minutes per graph. A similar factor can be observed in the automata size, see (b), meaning that not only do the runtimes increase drastically, but the automata also become larger. At 10 nodes with 4 sources, there are a whopping 100 million rules in an automaton. This size of the automata would present a challenge to any downstream application. At the same time, having access to more sources is crucial for coverage, as Figure 4.13(c) shows. At 10 nodes, only about one third of the graphs can be parsed with two sources. As mentioned before, Courcelle and Engelfriet (2012) show that with k sources, one can parse graphs with *tree width* $k-1$. Tree width is a measure of the complexity



(a) Runtimes

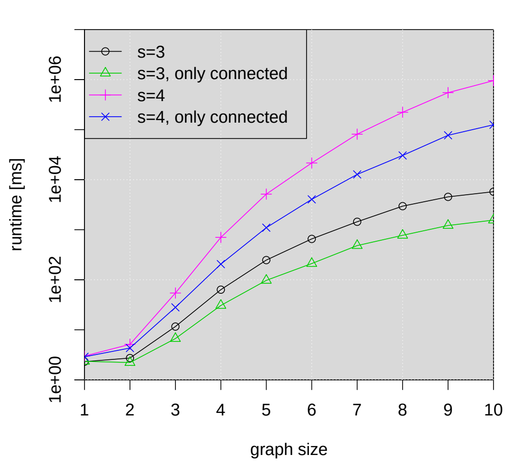


(b) Automata size

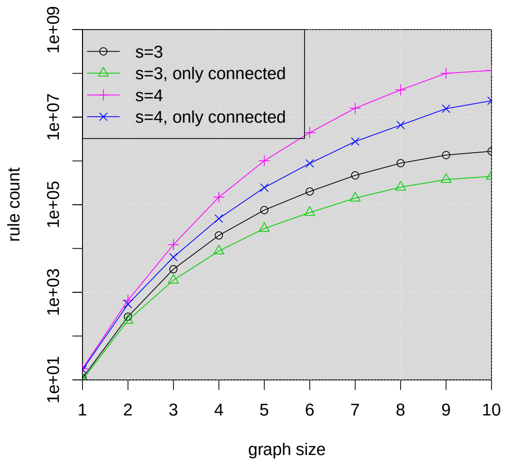


(c) Coverage

Figure 4.13: Experiment results for 2,3 and 4 sources, with the bottom-up algorithm

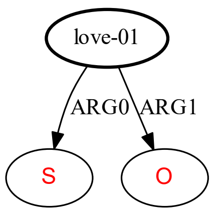


(a) Runtimes

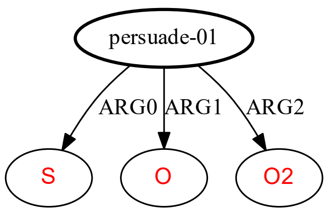


(b) Automata size

Figure 4.14: Restricting the bottom-up algorithm to only allow connected subgraphs reduces runtimes and automata size.



(a) A constant G_{love}

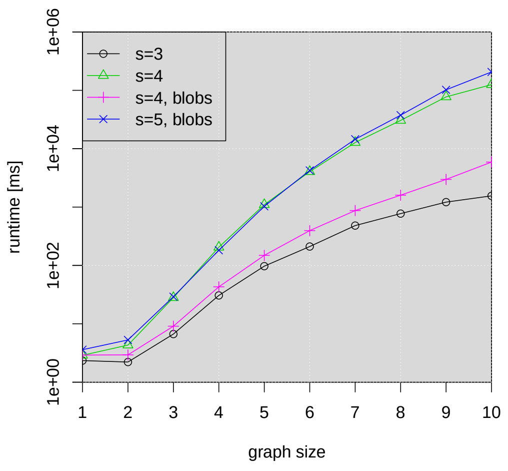


(b) A constant G_{persuade}

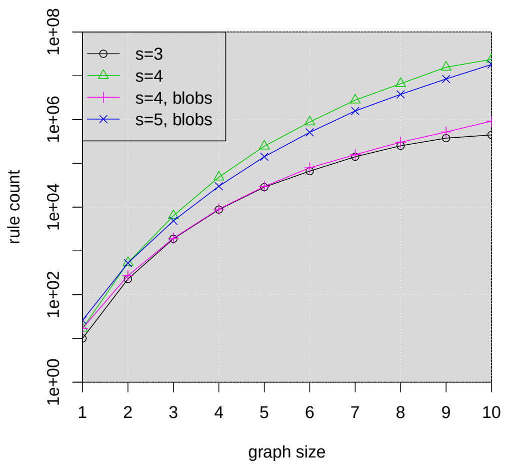
Figure 4.15: Larger 'blob' constants, as used in the IRTG in Chapter 3.

of a graph. For two sources, this means tree width 1, which corresponds to trees, i.e. graphs without reentrancies. Thus, the coverage in (c) for 2 sources shows the percentage of trees in the corpus. With three sources, the coverage jumps up to about 90%, and with four sources to nearly full coverage on this dataset.

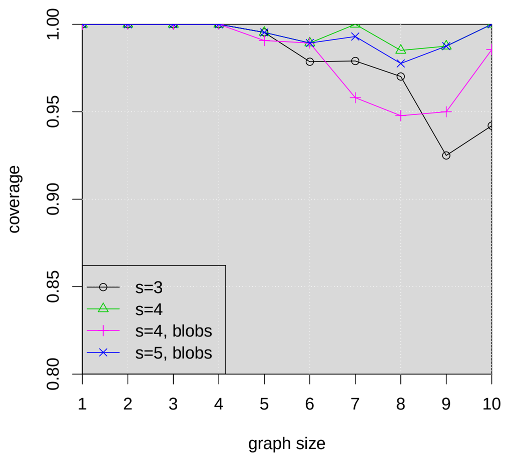
Restricting to connected subgraphs. One idea to reduce the complexity of these algorithms is to only allow connected subgraphs as states of the tree automata, similar to how most string parsers only allow consecutive spans as partial results. This removes some terms from the language of the automata, but the assumption is a reasonable one. Disconnected subgraphs are unlikely to form meaningful subunits in AMR. The performance increases, shown in Figure 4.14(a) are sizable, nearly an order of magnitude for four sources. A similar decrease in size can be observed for the automata, see (b).



(a) Runtimes



(b) Automata size



(c) Coverage

Figure 4.16: Comparing atomic constants and 3/4 sources to blob constants and 4/5 sources.

Larger constants. The IRTGs we saw in Chapter 3, based on Koller (2015), as well as the HRG grammar of Peng et al. (2015), use larger constants with some edges attached to nodes, such as in Figure 4.15. We experiment with this style of constant here. We attach to a node all outgoing edges with ARGx, opx, sntx ($x \in \mathbb{N}$), domain, poss or part labels, and all incoming edges with any other label. This heuristic closely matches the heuristic of Peng et al. (2015), with minor differences. This method partitions the edge set of the graph. The constants each have one labeled node, which we give the root source R, and unlabeled nodes, to which we assign arbitrary other source names from our fixed sets.
These constants make sense, they form meaningful units and reflect the idea of 'argument slots' established in Chapter 3. However, they also need a lot of sources just to write down. A graph like G_{love} in Figure 4.15, for a transitive verb, needs three sources (including the root), and G_{persuade} already needs four. The empirical results in Figure 4.16(c) indicate that using the blob constants requires adding one source to achieve similar coverage. However, as (a) in the same figure shows, this trade-off does not reduce runtimes after all, nor does it significantly reduce automata sizes, as (b) indicates. We will thus stick with the atomic single edge and single node constants for the rest of this chapter. However, the blob constants will make a comeback with the AM algebra in Chapter 5.

Language size. The language of the HR decomposition automata is always infinite: since renames just swap sources, when one applies the same rename operation twice, it just cancels out. Thus, rename operations can occur in arbitrarily long chains at any point in the term, without changing the result. This leads to an infinite set of terms evaluating to the same graph. However, we can make an ad-hoc restriction to make the languages finite, and measure their size. We do this by annotating the automata states with information about which nodes have been involved in a rename operation since the last forget or merge. We then prohibit renaming the source on one node twice, without a forget or merge operation in between. This breaks the rename cycles, and makes the languages finite. Figure 4.17

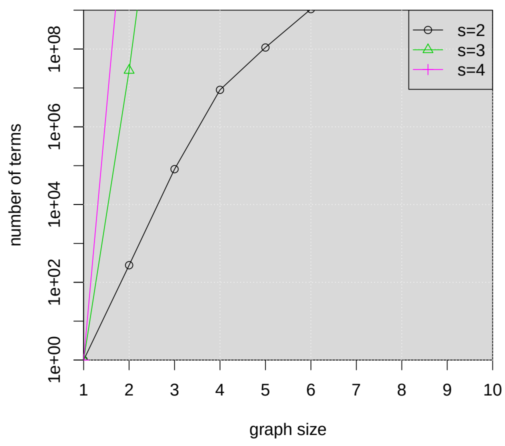


Figure 4.17: Language sizes of the restricted bottom-up automata.

plots the language sizes. There are astonishingly many terms for even very small graphs. For example, with three sources, even AMRs with just two nodes have over 10 million terms describing them. The plot caps at one billion terms, larger language sizes couldn't be measured for technical reasons.

Realistic data. Figure 4.18 shows results on the LDC2015E86 dataset. The plot uses the bottom-up algorithm, restricted to connected subgraphs and using the atomic constants, for 2, 3 and 4 sources. The plots reveal severe problems with this approach. Coverage with two sources is unsatisfactory – it is clear that we must do better than just parsing trees. With three sources, coverage is much better, but still not perfect. At the same time, automaton sizes get impractically large, and runtimes are a real concern. In this experiment, the full corpus couldn't be covered for 3 or 4 sources, despite running the experiment for several days. These problems are even more pronounced for four sources. We will address these issues in the next chapter, introducing the AM algebra that improves the situation dramatically.

4.7 Application: Graph parsing

A further application of this chapter's decomposition automata is *graph parsing*, i.e. using the graph as the input for the parsing process of a grammar. If the grammar is synchronous – as in the IRTG or synchronous HRG formalisms we discussed in we discussed in Section 3.1 –, then this parsing process yields a string as output, i.e. we could generate sentences from AMRs.

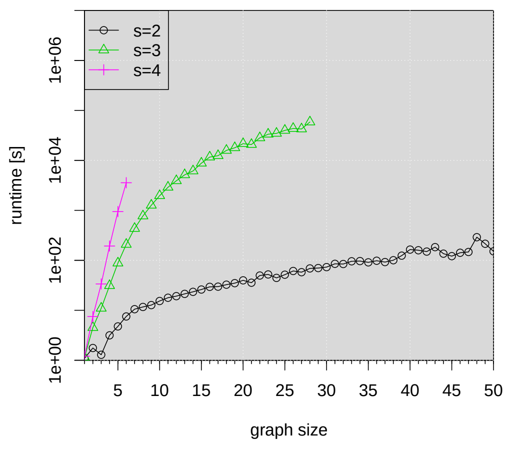
Groschwitz et al. (2015) demonstrates the feasibility of this application in the context of IRTG. Recall from Section 3.1.1 that to parse an object, we compute the parse chart

$$A_G \cap h_s^{-1}(D)\,,$$

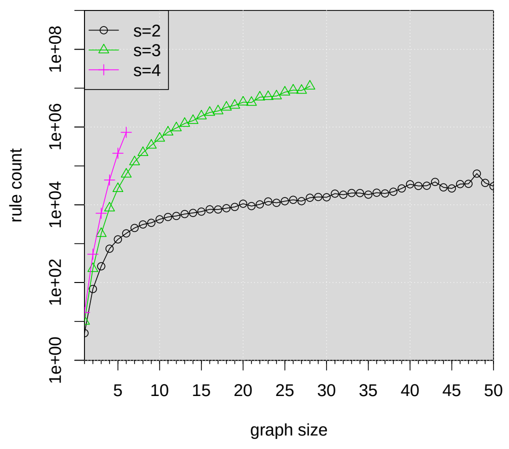
the intersection of the IRTG's grammar automaton A_G and the inverse homomorphism h_s^{-1} of the decomposition automaton D. The mechanisms for intersection and inverse homomorphism of tree automata are very general, but the decomposition automaton is a crucial ingredient for specifically the graph parsing process.

In fact, Teichmann et al. (2018) show that an IRTG can be brought into a normal form where the asymptotic runtime complexity of the parsing algorithm is determined by the complexity of computing the decomposition automaton. In other words, the runtime complexity of $O\left(n^s 3^{ds} ds\right)$ we showed in this chapter also holds for graph parsing with IRTG.

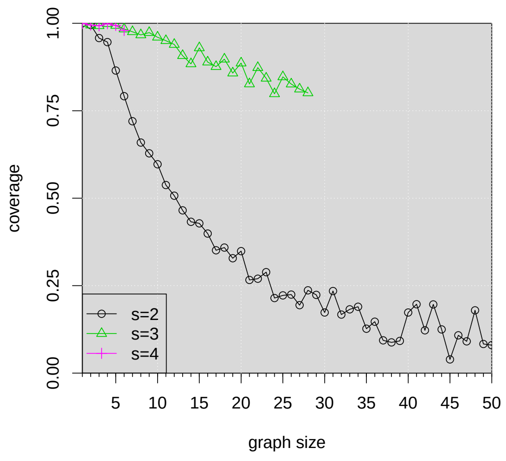
This is interesting because in related work, Chiang et al. (2013) show that graph



(a) Runtimes

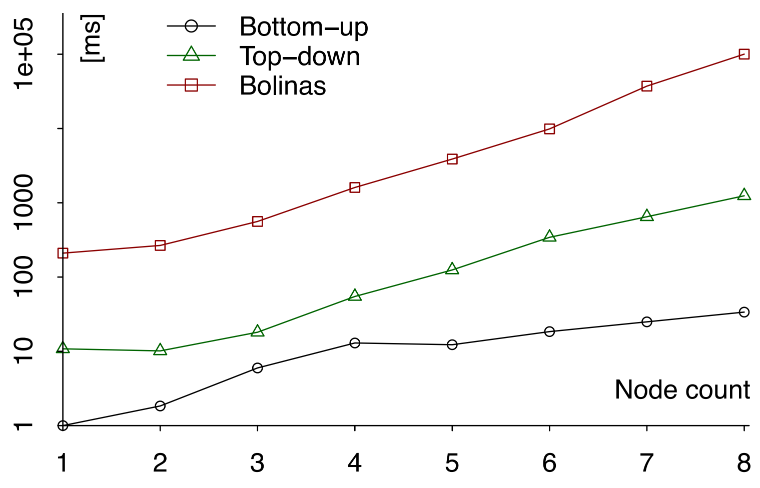


(b) Automata size

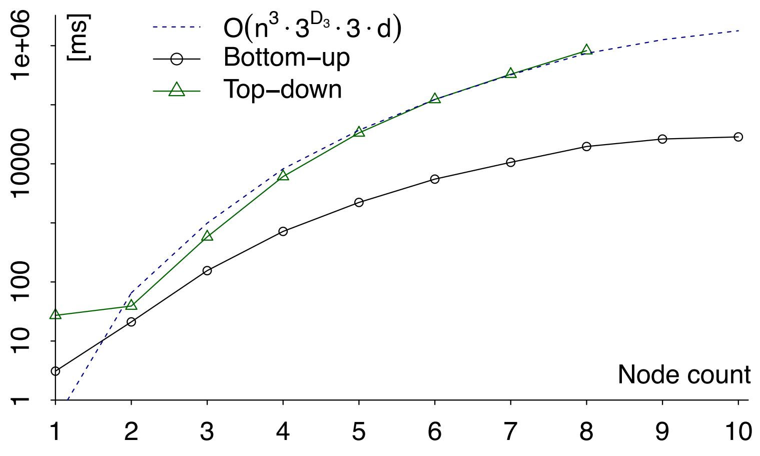


(c) Coverage

Figure 4.18: Evaluation on the LDC2015E86 dataset. This uses the bottom-up algorithm, restricted to connected subgraphs and using atomic constants.



(a) Comparison of the Groschwitz et al. (2015) system with Bolinas on a grammar with 2 sources, i.e. with treewidth 1.



(b) The Groschwitz et al. (2015) graph parsing system with 3 sources.

Figure 4.19: The runtime comparisons of Groschwitz et al. (2015).

parsing with HRG has runtime complexity $O\left(n^{k+1}3^{d(k+1)}d(k+1)\right)$, where k is a parameter of the grammar called its *treewidth*.[6] Groschwitz et al. (2015) extends the equivalence results between the HR grammar and HRG of Courcelle and Engelfriet (2012) by showing that an IRTG with an HR algebra with s sources is equivalent to an HRG with treewidth $k+1$. In other words, the complexities $O\left(n^s 3^{ds} ds\right)$ and $O\left(n^{k+1}3^{d(k+1)}d(k+1)\right)$ match exactly.

While Groschwitz et al. (2015) does not work with a synchronous grammar, the paper uses an IRTG with just an HR interpretation and provides empirical results on runtimes for computing the parse chart from an AMR. This process is also the bottleneck for parsing with a synchronous grammar, so the runtime results can be assumed to generalize. Figure 4.19 shows the paper's comparison with Bolinas[7], an implementation of the algorithm in Chiang et al. (2013), using equivalent grammars for both. The graph shows drastic practical runtime improvements, which were further improved in Groschwitz et al. (2016) where we describe faster algorithms for tree automata inverse homomorphism and intersection.

4.8 Conclusion

In this chapter, we described algorithms for computing decomposition automata for graphs with respect to the HR algebra, both in bottom-up and top-down fashion. We examined runtimes both in theory and in practice on an AMR corpus, and also looked at the application of graph parsing, where the runtime performance of this approach compares well to related work.

While we showed that computing the automata is feasible in practice, some problems remain if we want to use them to sample a consistent set of terms for training a parser. Chief among these problems is the fact that the resulting automata are huge, with millions of rules even for small graphs. And even with an ad-hoc restriction to make the set of terms finite, there are still billions of terms describing a single graph with just a couple of nodes.

In particular, we observed a trade-off in terms of source count: a low source count of two yields smaller automata and better runtimes, but has lower coverage and is unable to model important phenomena such as control. A high source count of four has unfeasible large automata and runtimes. And for three sources, instead of being the best of both worlds, both types of problems persist.

[6]In the paper, Chiang et al. (2013) list $O\left(n^{k+1}3^{d(k+1)}\right)$ as their runtime, but they ignore the per-rule runtime of $O\left(d(k+1)\right)$.

[7]`https://www.isi.edu/licensed-sw/bolinas/`

In the next chapter, we will have a closer look at why there are so many terms over the HR algebra, and why this makes it hard to generate a consistent set of terms for training a parser. We will then introduce the AM algebra as a linguistically motivated alternative.

5

The AM Algebra

In the last chapter, we saw that for any single AMR, there are enormously many terms over the HR algebra, and even the decomposition automata as compact representations of the terms are huge and take long to compute. In this chapter, we investigate the reasons for this 'compositional complexity' and why it is a problem when generating terms to train a parser.

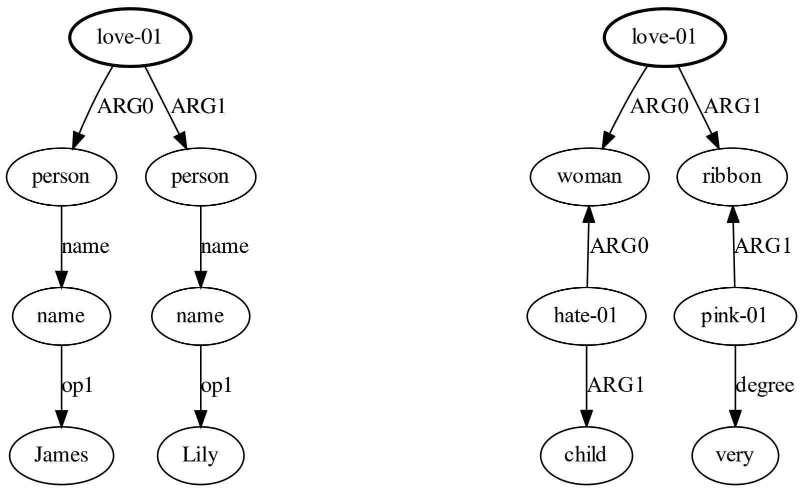
To resolve this problem, we introduce the Apply-Modify algebra (AM algebra), a graph algebra based on the linguistic principles discussed in Chapter 3. The AM algebra has its technical foundation in the HR algebra, but has a much lower compositional complexity: we find that for any given AMR there are drastically fewer terms over the AM algebra than over the HR algebra, and that the decomposition automata are much smaller and faster to compute.

We first discuss the problems with the HR algebra in Section 5.1, before we define and formally examine the AM algebra in Sections 5.2 and 5.3. In Section 5.4 we discuss the capabilities and limitations of the AM algebra in modeling different linguistic phenomena. Finally, Sections 5.5 and 5.6 show how to compute decomposition automata for the AM algebra in practice.

5.1 Problem statement

In the last chapter, we saw that the decomposition automata for the HR algebra are very big (millions of rules per AMR), take long to compute, and have a language that is either infinite or, with an ad-hoc restriction, finite but huge (billions of terms even for very small AMRs). Let us call this combination of automata size, computation time and language size loosely the 'compositional complexity' of an algebra. In other words then, we found that the HR algebra has a high compositional complexity.

We started Chapter 4 with the plan of using the decomposition automata of

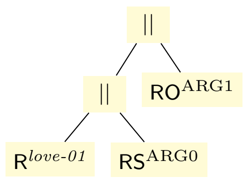


(a) *James loves Lily.* **(b)** *The woman who hates children loves very pink ribbons.*

Figure 5.1: Two example AMRs that are structurally similar.

the HR algebra to sample terms for grammar induction, in the style of e.g. Peng et al. (2015). We will now examine why the high compositional complexity of the HR algebra is a problem for the sampling process, and what the causes of this high complexity are. Recall that the idea of such a sampling-based method is to randomly generate a term such as the one in Figure 5.2(a) for a given AMR, and divide it into contiguous segments that can serve as HR terms in grammar rules. Example segments are indicated with the different colors in Figure 5.2(a).

In fact, these colored term segments correspond to the rules in the IRTG of Figure 3.1 (here reprinted in Figures 5.3 and 5.4 for convenience). The only difference is that the IRTG uses the large constants of Figure 5.4, whereas the term in Figure 5.2(a) uses the constants made of single nodes or edges (we saw in the last chapter that using the larger constants does not help with the issue, since they require us to use more source names). For example, the subterm



of the yellow segment evaluates to the graph G_{love} in Figure 5.4 – recall from Section 2.4 that e.g. $\mathsf{R}^{\textit{love-01}}$ denotes a single node with label *love-01* and source R, while $\mathsf{RS}^{\text{ARG0}}$ denotes an edge with label ARG0 from an unlabeled node with source R to an unlabeled node with source S. Thus, the yellow segment corresponds to the love rule, the blue to the $\mathsf{apply\_subj}$ rule and the red and green segments to

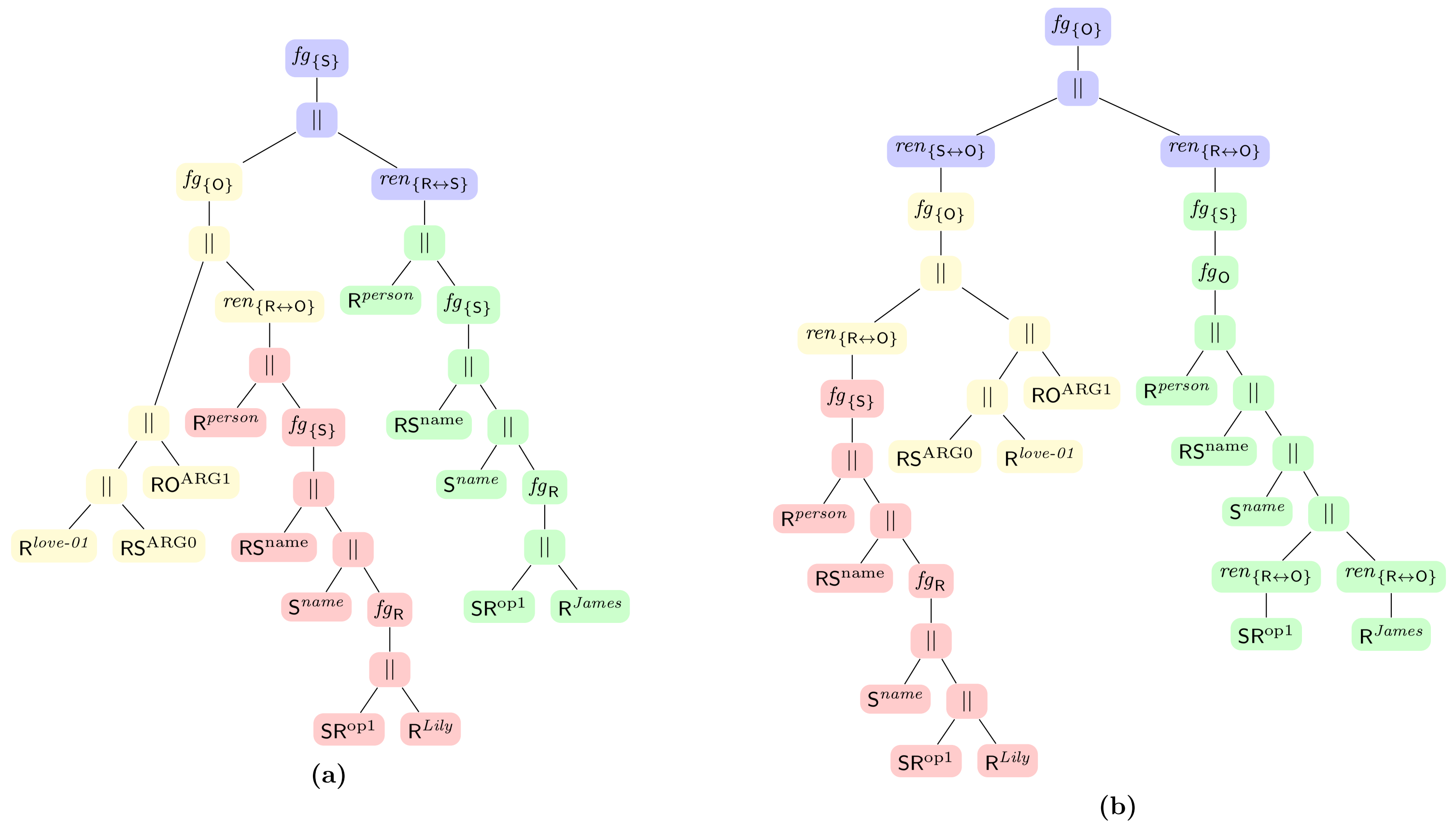


Figure 5.2: Two terms with functionally equivalent term segments colored.

RTG rules	graph homomorphism h_g	string homomorphism h_s
NP $\rightarrow$ james	G_{James}	James
NP $\rightarrow$ lily	G_{Lily}	Lily
VP $\rightarrow$ love (NP)	$fg_{\mathsf{O}} \left(G_{\text{love}} \mid\mid ren_{\mathsf{R} \mapsto \mathsf{O}} \left(x_1 \right) \right)$	$\mathsf{loves} * x_1$
S $\rightarrow$ $\mathsf{apply\_subj}$ (NP, VP)	$fg_{\mathsf{S}} \left(x_2 \mid\mid ren_{\mathsf{R} \mapsto \mathsf{S}} \left(x_1 \right) \right)$	$x_1 * x_2$

Figure 5.3: A small handwritten IRTG that can analyze the sentence *James likes Lily*. The graph constants G_{X} are shown in Figure 5.4. The start symbol of the RTG is S.

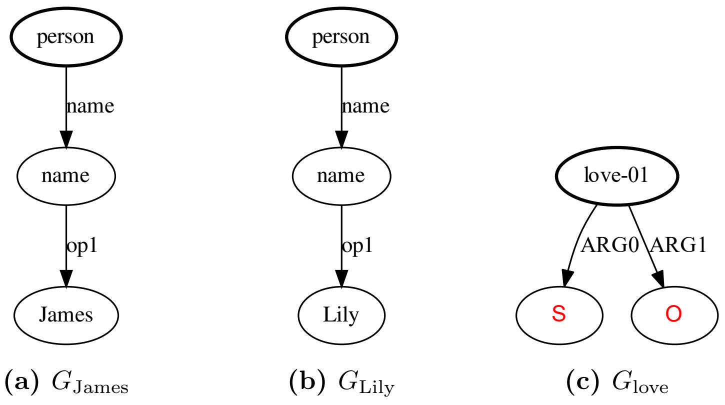


Figure 5.4: The graphs referenced in Figure 5.3.

the lily and james rules respectively.

During the sampling process, each sampled term segment influences the probabilities in future sampling iterations across the corpus, increasing the probability of sampling similar segments in the future. When this works well, the last round of sampled terms share many similar segments that can be used for rules that generalize well.

However, this was not the case when my colleague Christoph Teichmann ran preliminary sampling experiments with the HR decomposition automata of the last chapter (personal communication, 2016). These experiments used variations of the Chinese restaurant process and Gibbs sampling on the LDC2015E86 dataset to find repeating term segments. The sampling algorithm could not find enough patterns in the sampled segments to converge on a consistent set of rules, instead generating nearly completely different segments with each sample.

The challenge here lies with the high compositional complexity of the HR algebra. On the one hand, term segments that *do* the same thing don't necessarily *look* the same on the surface. For the two terms in Figure 5.2, the parts with the same colors are functionally equivalent, that is, at each of the 'cut points' between the colored segments, the partial results are the same. Figure 5.5 shows the corresponding

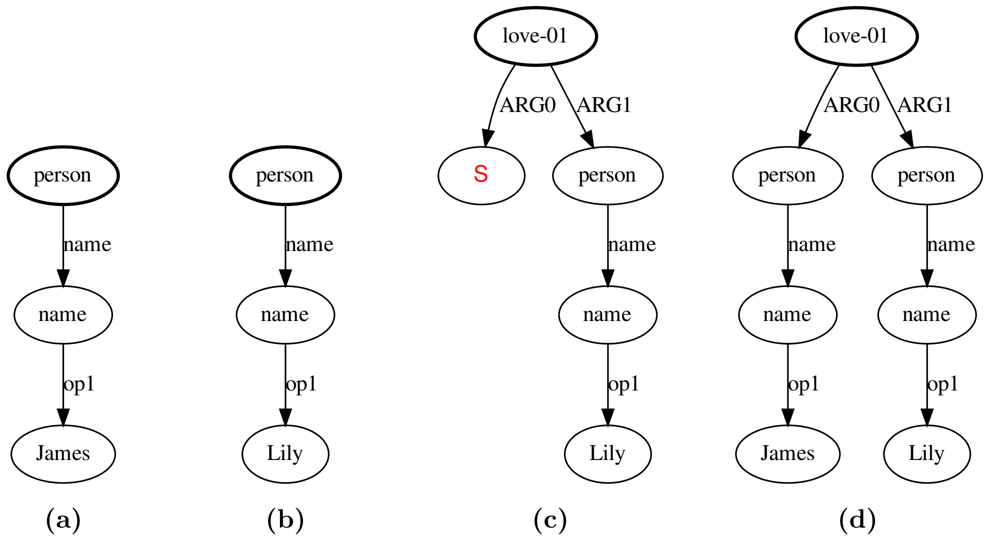


Figure 5.5: Partial results of the terms in Figure 5.2. Results are after the graph segment in (a) green, (b) red, (c) yellow, (d) blue.

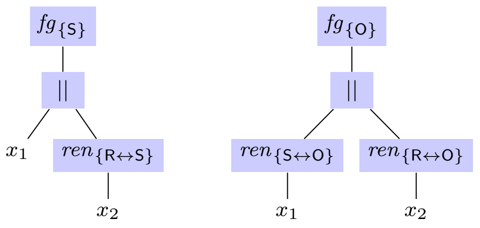

$fg_{\{\mathsf{S}\}}$
$||$
x_1
$ren_{\{\mathsf{R}\leftrightarrow\mathsf{S}\}}$
x_2
$fg_{\{\mathsf{O}\}}$
$||$
$ren_{\{\mathsf{S}\leftrightarrow\mathsf{O}\}}$
$ren_{\{\mathsf{R}\leftrightarrow\mathsf{O}\}}$
x_1
x_2

Figure 5.6: Two contexts that are functionally equivalent, as long as x_1 has exactly sources R and S, and x_2 just R.

graphs. In fact, the functional equivalence goes so far that for e.g. both blue contexts in Figure 5.6, the results are the same for *any* s-graphs entered into x_1 and x_2, as long as x_1 has exactly sources R and S, and x_2 just R. This condition can be guaranteed by the nonterminals of the IRTG. Thus, the two blue terms would effectively define the same IRTG rule; and similarly for the other colors. And yet, the term segments themselves are not identical. This **surface ambiguity**, meaning that functionally equivalent term segments can be different on their surface, makes it hard for the sampling algorithm to find patterns: there may be many cases where we find segments that are functionally equivalent, but the sampler does not recognize them as such. There are many causes of surface ambiguity in the HR algebra:

- The merge operation is symmetric, i.e. the order of arguments can be swapped. The merges of $\mathsf{R}^{\textit{love-01}}$ and $\mathsf{RS}^{\text{ARG0}}$ in the yellow segments are an example of this. Further, the order of consecutive merges can often be swapped.
- Forget operations can often change order with other operations (see e.g. the $fg_{\{\mathsf{S}\}}$ operation in the red segment).
- Spurious renames can occur nearly everywhere. In fact, without the ad-hoc restriction on consecutive renames we introduced in the last chapter, infinite rename chains and thus infinite variations on the graph segments are possible. But even when controlling consecutive renames, source names can be switched around spuriously. For example, the blue segment in Figure 5.2(b) uses the O source instead of the S source to merge the segments, with the same result; something similar occurs in the green segment as well.

Typically, such surface ambiguities are addressed with normal forms, where for different equivalent surface structures, a default one is chosen. This is one aspect of the AM algebra we present as a solution below.

On the other hand, there is also a **functional ambiguity** in the HR terms. There are many different ways to divide a single term into segments, and there are many terms for a single graph. This means that even for a single graph, there is a gigantic number of term segments we can sample, even aside from surface ambiguity.

Consider for example the term in Figure 5.7(a), which also generates the graph in Figure 5.1(a). Here, the purple segment adds the ARG0 edge to the *love-01* node, including the attached *person* and *name* nodes and the name edge, but not yet the actual name *James*. The teal segment then adds the same parts for the ARG1 edge of *love-01*. The result is shown in Figure 5.8(a). That is, first a part of the left side of the AMR in Figure 5.1(a) is added, then a part of the right side. Only then is the name *James* added on the left of the AMR (gray segment in Figure 5.7(a)), and

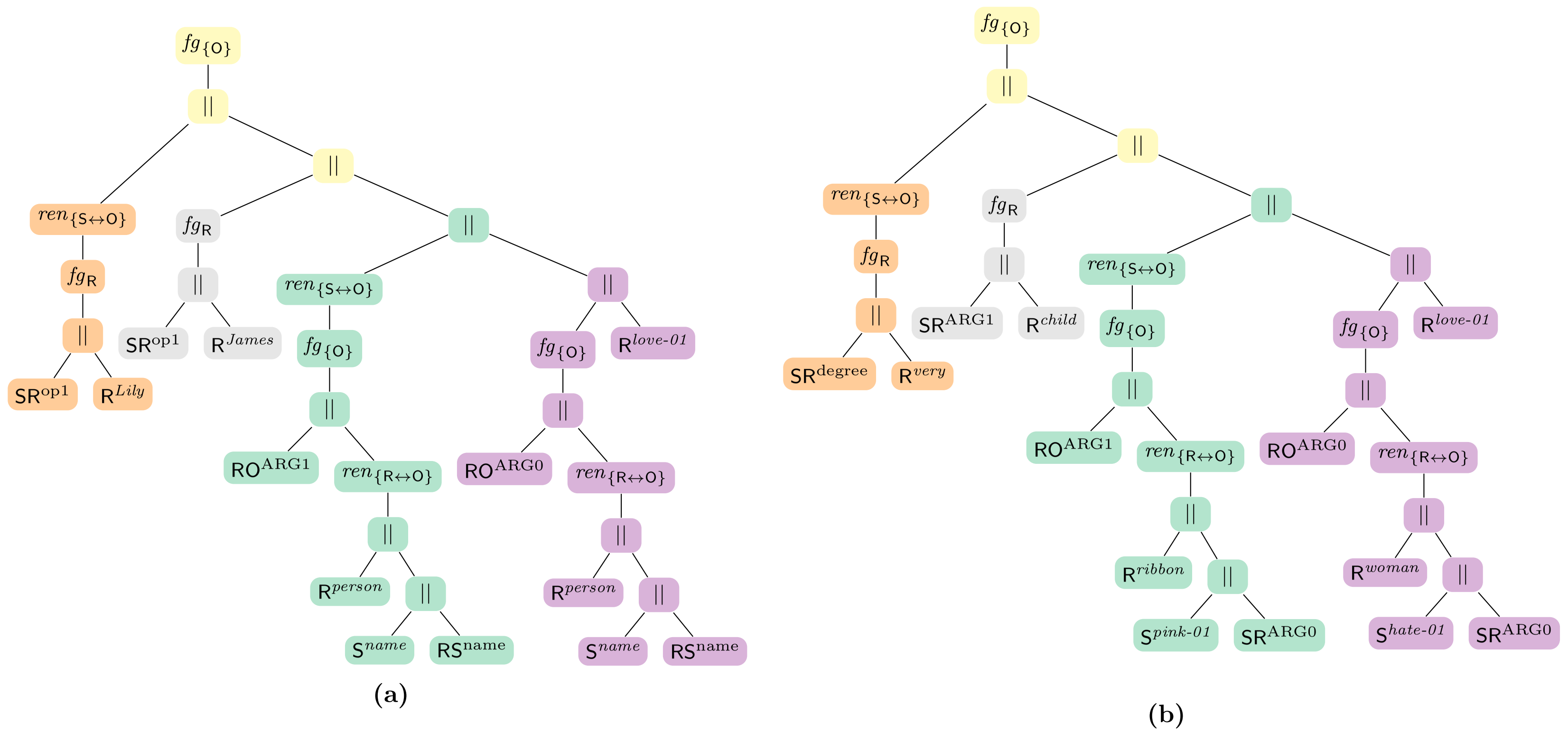
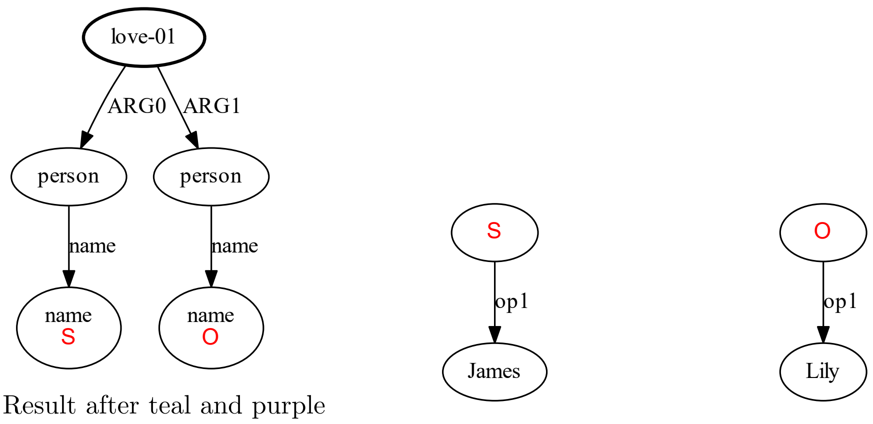


Figure 5.7: (a) A term structurally different from the ones in Figure 5.2, and (b) a similar term for the graph in Figure 5.1(b)



(a) Result after teal and purple subterms. **(b)** After gray subterm. **(c)** After orange subterm.

Figure 5.8: Partial results of the term in Figure 5.7(a).

then is *Lily* added on the right side of the graph (orange in the term). None of the segments in Figure 5.7(a) is functionally equivalent to a segment in Figure 5.2(a). (In fact, due to the alternating pattern, the term in Figure 5.7(a) *cannot* be divided into contiguous segments that are functionally equivalent to the ones marked in Figure 5.2(a)).

This sort of functional ambiguity sounds on the one hand like the exact thing where we would want to offer the sampling algorithm all possibilities, such that it can find the most useful term segments itself. That is, if we do not have prior knowledge about how to choose among the different terms and the functionally different segments, the sampling procedure should choose the segments that explain the data best, i.e. segments that encode repeating patterns. But when decomposing AMRs with the HR algebra, there are *too* many functionally different terms and segmentations for a single graph. When sampling from them without strong prior assumptions, we won't find repeating segments very often. Fortunately, we *do* have some prior knowledge about which of these terms and segments are desirable, and which are not.

For example, in the graph in Figure 5.1(a), clearly the subgraphs corresponding to G_{James} and G_{Lily} (c.f. Figure 5.4) are meaningful units and should be added as arguments of *love-01* as a whole. In the term in Figure 5.7(a) however, these subgraphs are added each in two separate, non-contiguous term segments (purple and gray, and teal and orange respectively). In other words, the purple segment adds the *person* and *name* nodes of G_{James} as the subject of *love-01* before the meaningful subgraph G_{James} is complete. One may now object that G_{James} as a named entity is a special case, and could e.g. be condensed into a single node in a preprocessing step. However, the same phenomenon occurs for the AMR in Figure 5.1(b), corresponding to the sentence *The woman who hates kids loves very pink ribbons.* This AMR is structurally the same (except for edge directions), and a term analogous to Figure 5.7(a) is shown in Figure 5.7(b). Again, parts of the phrases are added in alternating order, first adding the *hate-01* and *woman* nodes, then the *pink-01* and *ribbon* nodes and only then are the nodes for *child* and *very* added. That is, again, the object and subject of *love-01* are added before the meaningful subgraphs are complete. It would be desirable to instead build meaningful subgraphs in single contiguous terms. The challenge in reducing functional ambiguity lies in the fact that while we have some prior knowledge, we are far from knowing everything about language. The question is thus, how do we encode that which we know, while giving the statistical model enough leeway to learn what we don't know?

As a separate issue, **long runtimes** for computing the HR decomposition automata are a real concern for sampling. While the empirical performance we saw in the last chapter is reasonable on the smaller graphs of the Little Prince corpus, on the larger LDC2015E86 corpus computation became memory and time intensive, to the point of infeasibility.

All these problems get amplified when using more source names. In fact, as the last chapter demonstrated, this effect is rather drastic. Even at just three sources, automata size and computation time become unpractical, and for four source names, automata become incredibly large and took infeasible amounts of resources to compute. At the same time, however, it seems like at least four sources are needed to achieve satisfying coverage.

Looking at the terms in Figure 5.2, one might think that using larger constants would resolve many of the problems. However, as we saw in the last chapter, larger constants require more sources to achieve reasonable coverage. Empirically, this turned out not to be worth it as far as the trade-off of coverage and compositional complexity is concerned.

In summary, the HR algebra's high compositional complexity presents a challenge for statistical grammar induction via sampling. While there may be technical solutions to this, we take a different approach here. The insight is the following. The HR algebra is a general-purpose algebra for building essentially any graph. But AMRs are not just nodes and edges, they are *semantic* graphs, and we saw in Section 3.2 how they can be constructed with the methods of application, modification and unification, which are more specific to semantic construction. We saw that the HR algebra can be used to model these operations, so that is not the problem. The problem is that the HR algebra can also do everything else. So how about an algebra that can perform *exactly* application, modification and unification? This is the idea behind the AM algebra presented in this chapter.

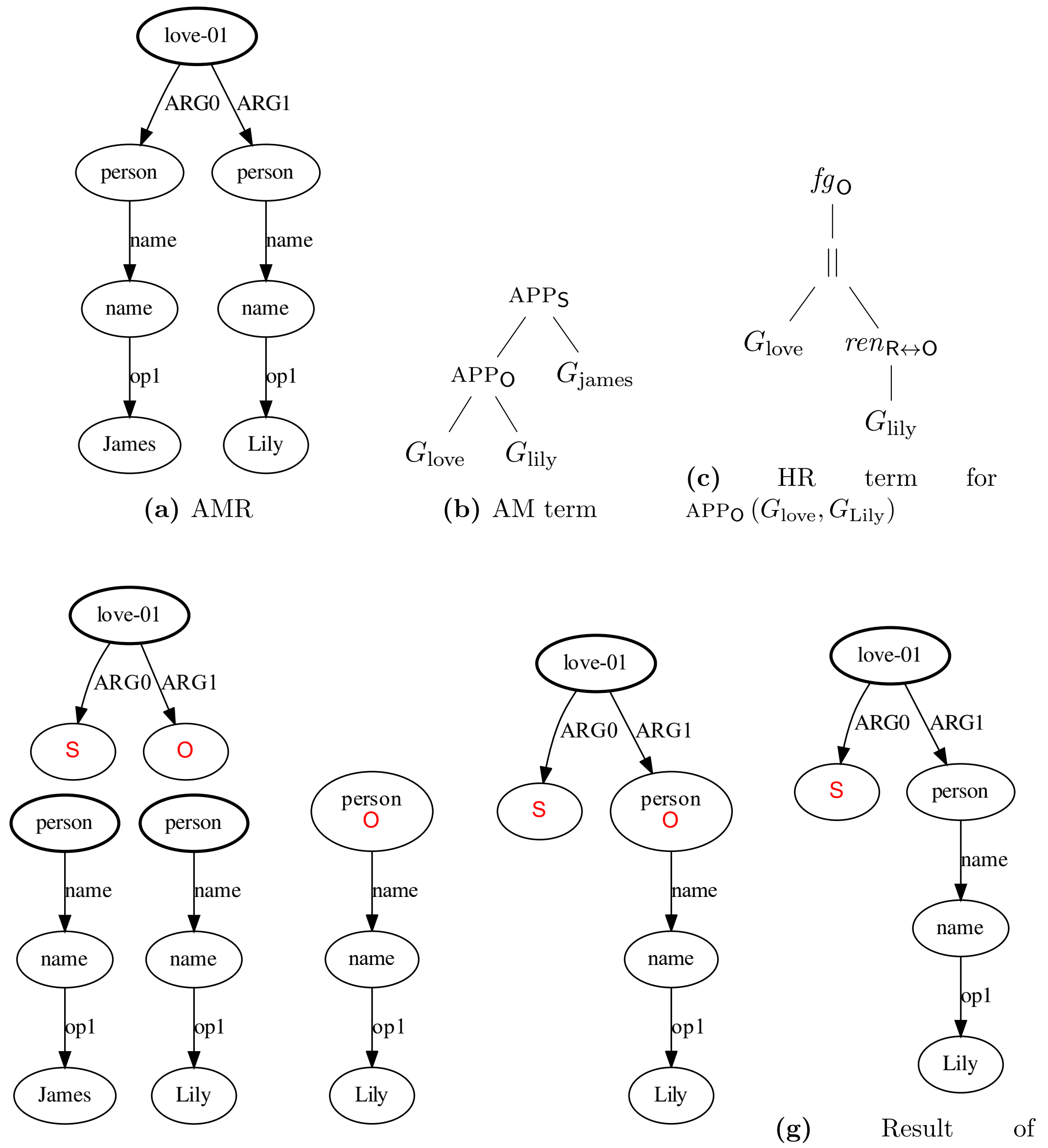
Related work on normal forms. Ambiguities similar to the ones discussed here also occur elsewhere in Computational Linguistics. *Spurious ambiguities*, for example, occur when multiple syntactic derivations are semantically equivalent. If spurious ambiguity is high, this can increase parsing cost drastically. Eisner (1996) gives a precise definition of semantic equivalence in the context of CCG, and presents a normal form algorithm that eliminates all spurious ambiguity for a specific version of CCG. Hoyt and Baldridge (2008) present a reformulation of the approach, and Hockenmaier and Bisk (2010) extend it to include e.g. type raising.

While both the approach of Eisner (1996) to spurious ambiguity and the AM algebra presented here reduce ambiguities in term structures, there are major differences. Spurious ambiguity concerns different syntactic derivations that yield the same meaning, whereas here we have the 'meaning' given in the form of an AMR and consider ambiguity on the level of fragments of semantic terms. Further, we do not only want to condense equivalent term fragments into one (reduce surface ambiguity) but also rule out undesirable term fragments (reduce functional ambiguity). Finally, Eisner (1996) reduces spurious ambiguity by restricting when which rules can be applied. While this is also a part of the AM algebra (due to its type system), a large part of the reduction in ambiguity is because the AM algebra combines multiple HR operations into a single operation.

5.2 The AM algebra

In this section, we define the *apply-modify (AM) algebra*, that has two types of operations, application and modification. Each combines a sequence of HR operations into one higher level operation. Before we define the application and modification operations formally below, let us have a look at some examples.

Application We first look at the *apply* operation APP_α, where α is a source name. Let us take a look again at our running example, *James loves Lily*, in Figure 5.9. The AMR in (a) is described by the term over the AM algebra in (b). The APP_{O} operation corresponds to the HR term in (c): first, the operation $ren_{\mathsf{R}\leftrightarrow\mathsf{O}}$ renames the root source R of G_{Lily} to O, the source in the slot we are filling; see Figure 5.2(e). Then, this result is merged with G_{love}, see (f), and the O source is forgotten, see (g). This effectively plugged the root of the *argument* G_{Lily} into the O slot of the *head* G_{love}. The APP_{S} operation similarly plugs the root of G_{James} into the S slot, to yield



(a) AMR

(b) AM term

(c) HR term for $\text{APP}_{\mathsf{O}}\,(G_{\text{love}}, G_{\text{Lily}})$

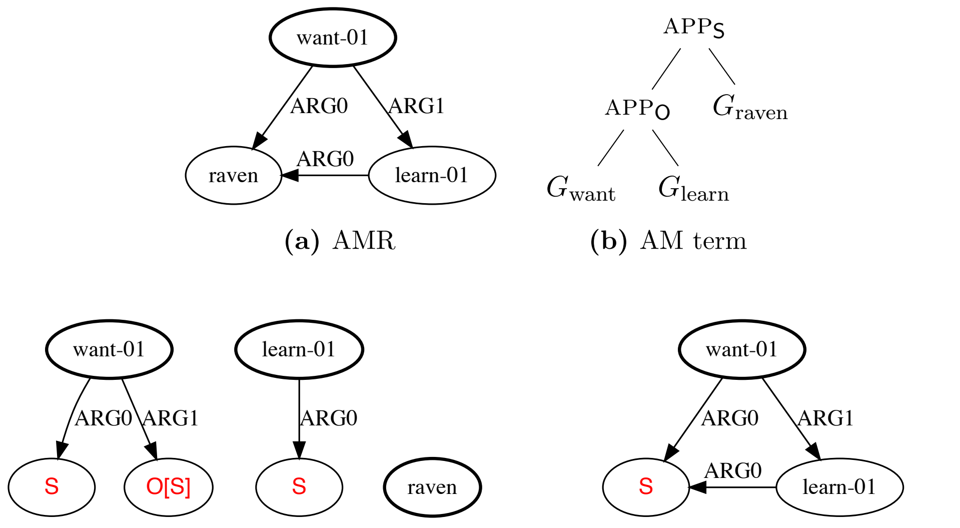
(d) Constants G_{love}, G_{James} and G_{Lily}

(e) Result of $ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\,(G_{\text{Lily}})$

(f) Result of $G_{\text{love}} \mid\mid ren_{\mathsf{R}\leftrightarrow\mathsf{O}}\,(G_{\text{Lily}})$

(g) Result of $\text{APP}_{\mathsf{O}}\,(G_{\text{love}}, G_{\text{Lily}})$, i.e. of the HR term in (c)

Figure 5.9: AMR and its analysis for *James loves Lily.*



(a) AMR **(b)** AM term

(c) Constants G_{want}, G_{learn} and G_{raven} **(d)** Result of APP$_O$ $(G_{\text{want}}, G_{\text{learn}})$

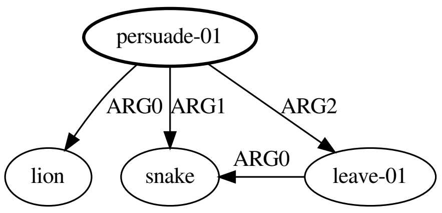
Figure 5.10: AMR and its analysis for *The raven wants to learn.*

the final AMR in (a). This pattern matches how we filled slots with the HR algebra before, for example the top of the yellow term segment or the whole blue segment in Figure 5.2(a); or in the IRTG in Figure 5.3 in the love and apply_subj rules.

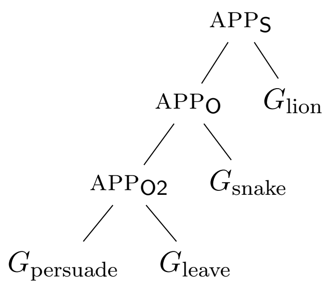
In this simple case, the graphs G_{Lily} and G_{James} were complete; their only source was R. However, in certain cases, we want to combine a predicate with an argument that is itself still looking for arguments. Take for example the graph in Figure 5.10(a), corresponding to the sentence *The raven wants to learn*, where, as we saw in Section 3.2, the raven is both the wanter and the learner. For the subject control verb *want*, we use the graph G_{want} of Figure 5.10(c). Its O source is *annotated* with the *argument type* [S] (written O[S]). This means that G_{want} requires its object argument to contain an S-source; during application this node is unified with the S-source of G_{want} itself, see Figure 5.10(d). We use the same HR term as in Figure 5.9(c) – just with different constants–, which makes the unification happen automatically in the HR merge operation. As seen in previous chapters, this unification yields an (undirected) cycle, a reentrancy. Note that due to the annotation O[S], this reentrancy was already *encoded in the constant* G_{want}. In fact, all reentrancies created by the AM algebra are encoded like this.[1] In Section 5.2.2, we discuss a more technical statement on how the AM algebra enforces that 'promise' of a reentrancy made by the constant.

The fact that annotations encode restrictions on the argument makes the AM

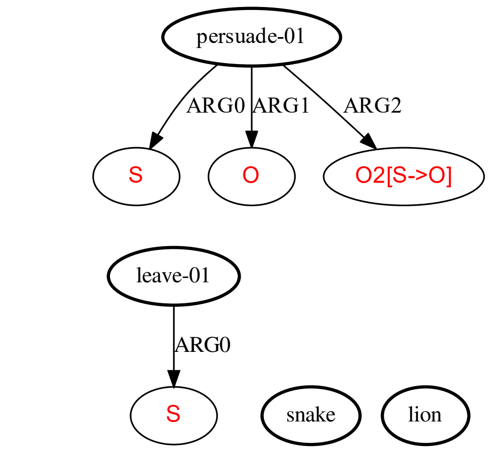
[1]See the paragraph on coreference in Section 5.4.1 for a discussion on a type of reentrancy that the AM algebra cannot encode well.



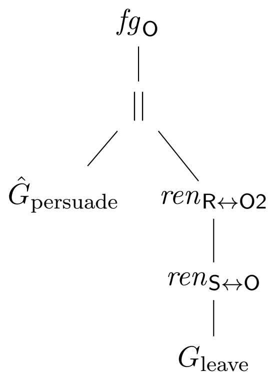
(a) AMR


APP_{S}
APP_{O}
G_{lion}
APP_{O2}
G_{snake}
G_{persuade}
G_{leave}

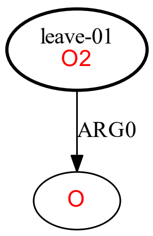
(b) AM term



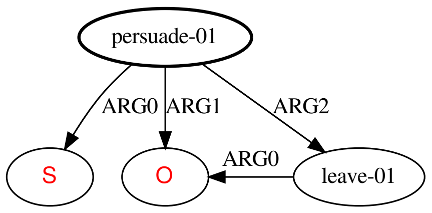
(c) Constants G_{persuade}, G_{leave}, G_{snake} and G_{lion}


fg_{O}
$||$
$\hat{G}_{\text{persuade}}$
$ren_{\mathsf{R}\leftrightarrow\mathsf{O2}}$
$ren_{\mathsf{S}\leftrightarrow\mathsf{O}}$
G_{leave}

(d) HR term for $\text{APP}_{\mathsf{O2}}(G_{\text{persuade}}, G_{\text{leave}})$. $\hat{G}_{\text{persuade}}$ is G_{persuade} without the $[\mathsf{S} \to \mathsf{O}]$ annotation.

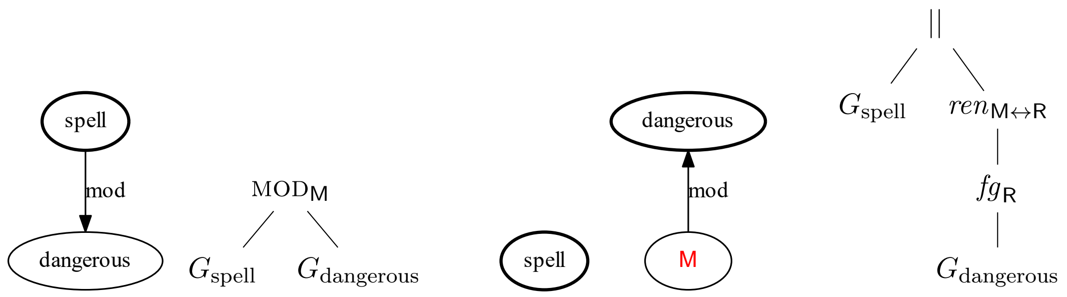


(e) Result of $ren_{\mathsf{R}\leftrightarrow\mathsf{O2}}(ren_{\mathsf{S}\leftrightarrow\mathsf{O}}(G_{\text{leave}}))$



(f) Result of $\text{APP}_{\mathsf{O2}}(G_{\text{persuade}}, G_{\text{leave}})$, i.e. of the HR term in (d)

Figure 5.11: AMR and its analysis for *The lion persuades the snake to leave.*



(a) AMR **(b)** AM term **(c)** Constants G_{spell}, $G_{\text{dangerous}}$ **(d)** HR term

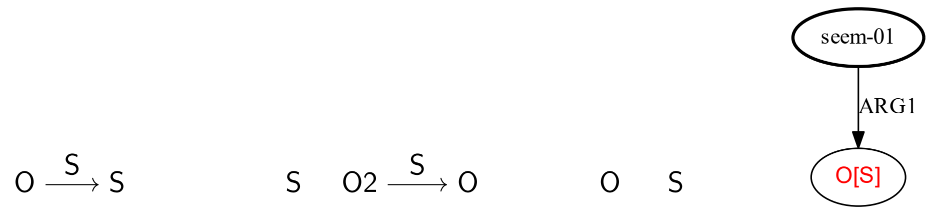
Figure 5.12: AMR and its analysis for *a dangerous spell.*

algebra a *partial* algebra, i.e. the operations are not always defined. For example, we cannot fill the O slot of G_{want} with G_{raven}, because G_{raven} does not have an S source. This allows the annotations to guide the composition process.

We also define annotations for *renaming* sources, in order to model phenomena such as object control verbs, as in *the lion persuaded the snake to leave*, see Figure 5.11. Here, the snake is both the leaver and the entity being persuaded. We can handle this with the graph G_{persuade} in Figure 5.11(c), that features an $\mathsf{O2}$-source which is annotated as $\mathsf{O2}[\mathsf{S} \rightarrow \mathsf{O}]$. This $\mathsf{O2}$ source must be filled by a graph that still has an S-source, which is renamed to an O-source during application, and thus merged with the O-source of persuade. This yields the structure shown in (f). The APP_{O2} operation here corresponds to an HR term with an additional rename operation $ren_{\mathsf{S}\leftrightarrow\mathsf{O}}$, as seen in (d). The result of the renames is shown in (e), and the merge operation again makes the unification happen automatically. While this renaming-annotation allows us flexibility in where reentrant edges should go, it is encoded in the constant. This not only makes sense because the object control is a property of the word *persuade*, but also binds all renaming operations to constants, preventing spurious renames.

Modification The modify operation is similar in nature to the apply operation, but is its inverse in terms of where the root ends up. A typical modifier is $G_{\text{dangerous}}$ of Figure 5.12(c), having an M source to be filled. In the operation $\text{MOD}_{\mathsf{M}}\,(G_{\text{spell}}, G_{\text{dangerous}})$, we first forget the root (i.e. the R-source) of the modifier $G_{\text{dangerous}}$, rename then its M source to R and merge the graphs, see Figure 5.12(d). This attaches $G_{\text{dangerous}}$ to the root of the head G_{spell} with the original M slot, see Figure 5.12(a) for the resulting AMR.

This is different to the apply operation in that if we were to evaluate $\text{APP}_{\mathsf{M}}\,(G_{\text{dangerous}}, G_{\text{spell}})$, the root R would be at the *dangerous* node, but here, the



(a) $\tau\left(G_{\text{want}}\right)$, $\tau\left(G_{\text{seem}}\right)$ **(b)** $\tau\left(G_{\text{persuade}}\right)$ **(c)** $\tau\left(G_{\text{love}}\right)$ **(d)** G_{seem}

Figure 5.13: Source annotation auxiliary graphs for (a) G_{want} of Figure 5.10(c), (b) G_{persuade} of Figure 5.11, (c) G_{love} of Figure 5.9. Also in (d) a graph G_{seem} that has the same type as shown in (a).

spell node remains the root. We can interpret this as G_{spell} remaining the head of the expression, being modified by the *adjunct* G_{spell}.

Note how the apply operation consumes a source of the head, whereas the modify operation does not do that, it leaves the sources of the head unchanged. Thus, in the AM algebra any head can be modified indefinitely often, mirroring the property of language we observed in Section 3.2.

We thus saw the ideas behind the application and modification operations of the AM algebra, and how source annotations control reentrancies. We can now define the algebra formally.

5.2.1 Formal definition

Let us start with the source annotations. We define the source annotations as one auxiliary graph per s-graph. For example, in Figure 5.13(a), the annotation $\tau\left(G_{\text{want}}\right)$ of the graph G_{want} of Figure 5.10(c) is shown. Its nodes are O and S, the sources of G_{want}. This is a shift of perspective, since in this auxiliary graph the source names do not play the role of extra labels that unify nodes, but are themselves the nodes. The arrow from O to S indicates that the O argument is required to have an S source, i.e. what we write as O[S] in the constant. The fact that the edge to S is also labeled S indicates that no rename occurs. It is worth pointing out that not all nodes in the source annotation must be sources in the graph, for example the graph G_{seem} in Figure 5.13(d) has the same type shown in (a), but no S source itself. Only later will the $\textsc{app}_{\mathsf{O}}$ operation introduce an S source, namely the S source of the argument. We will discuss this construction for raising verbs like *seem* below in Section 5.4.1.

Figure 5.13(b) shows the annotation $\tau\left(G_{\text{persuade}}\right)$ of the graph G_{persuade} of Figure 5.11(c). The S source is separate, indicating that the S source of G_{persuade} will not be part of any unification. The edge from O2 to O with label S indicates that

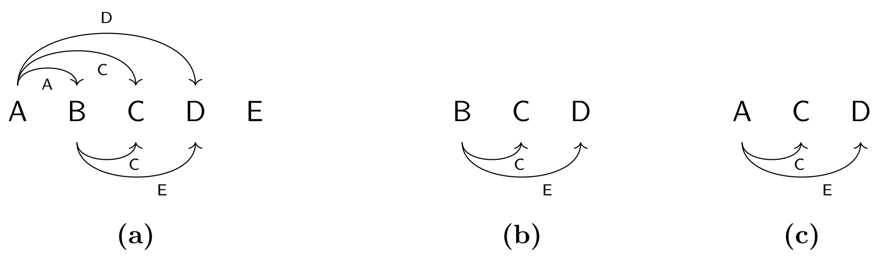


Figure 5.14: (a) A DAG that is also a source annotation, (b) the subgraph dominated by B and (c) the request at A

the O2 argument is required to have an S source, which will be renamed to (and thus unified with) O.

The graph in (c) is the annotation for G_{love} of Figure 5.9, where neither the O nor the S source is annotated. Thus, the annotation just contains the two sources, with no edges.

We call an s-graph paired with such an auxiliary graph an *annotated s-graph* or *as-graph*; a formal definition follows below. Defining the annotations through auxiliary graphs like this causes a bit of technical overhead at first, but will yield a clean and robust system in the end. Formally, we define the source annotations as directed acyclic graphs (DAGs), with the sources as nodes, no node labels, and sources as edge labels. First, the definition of DAGs.

Definition 5.1 (DAG)**.** A *directed acyclic graph*, or *DAG*, with edge labels in Λ is a tuple $G = (V_G, E_G, \lambda_G)$ where

(i) V_G is the set of nodes,

(ii) $E_G \subseteq V_G \times V_G$ is the set of directed edges,

(iii) $\lambda_G : E_G \to \Lambda$ labels each edge, and

(iv) there is no cycle, i.e. no directed path $v_0 \to_G^* v_k$ where $k \geq 1$ and $v_0 = v_k$.

Note that the last condition also excludes loops (with $k = 1$).

An example DAG is shown in Figure 5.14(a). We call a node v in a DAG G an *origin* in G if it has no incoming edges.[2] The origins in Figure 5.14(a) are A and E. For a DAG G and a node $v \in V_G$, we say the graph *dominated by* v is the subgraph induced by all nodes in G that can be reached from v, i.e. the nodes u to

[2]What we call "origin" here is often instead called the *root*, or *source* of the tree, but since this thesis uses the words "root" and "source" so much in different contexts, we use "origin" here.

which is a directed path from v (this includes v). The subgraph dominated by B in Figure 5.14(a) is shown in (b).

We use the following type of DAG for our source annotations, and for related notions later.

Definition 5.2 (Source annotation)**.** Let $\mathcal{S}$ be a set of sources. A *source annotation over* $\mathcal{S}$ is a directed acyclic graph (DAG) over $\mathcal{S}$ with edge labels in $\mathcal{S}$, i.e. a tuple $\tau = (V_\tau, E_\tau, \lambda_\tau)$ where

(i) $V_\tau \subseteq \mathcal{S}$, i.e. the nodes are sources,

(ii) the edge labeling function $\lambda_\tau : E_\tau \to \mathcal{S}$ labels each edge with a source,

such that additionally

(iii) every node has a direct edge to every node it dominates, i.e. if $v, u \in V_\tau$ such that $v \to^* u$, then $(v, u) \in E_G$, and

(iv) all edges leaving one node are uniquely labeled, i.e. for every node $v \in V_\tau$, for two edges $(v, u) \neq (v, w)$ we have $\lambda_\tau(v, u) \neq \lambda_\tau(v, w)$.

The DAG in Figure 5.14(a) is in fact a source annotation over $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$. Also, all the auxiliary graphs in Figure 5.13 are source annotations according to this definition.

Since it is rather inconvenient to draw the source annotations as graphs, given that they are often rather simple structures, we linearize them in the following way. Let τ be a source annotation.

First, we define a helper function L that maps each source α in τ to a string. If α is a leaf, we define $L(\alpha)$ as the string "α". If α is a non-leaf node with outgoing edges $e_1, \ldots, e_k$ to sources $\alpha_1, \ldots, \alpha_k$, we recursively define $L(\alpha)$ as the string

$$L(\alpha) = \alpha[\lambda_\tau(e_1) \to L(\alpha_1), \ldots, \lambda_\tau(e_k) \to L(\alpha_k)].$$

That is, for each outgoing edge, we add "edge label $\to$ string for edge target". To keep the notation as short as possible, we follow the convention that if the edge label equals the target node, i.e. $\lambda_\tau(e_i) = \alpha_i$, we write just $L(\alpha_i)$ instead of $\lambda_\tau(e_i) \to L(\alpha_i)$. Furthermore, we observe that if not only $\lambda_\tau(e_i) = \alpha_i$ but also there is a directed path from another α_j to α_i, then $L(\alpha_i)$ is already described in $L(\alpha_j)$. In this case, we omit $\lambda_\tau(e_i) \to L(\alpha_i)$ completely in $L(\alpha)$.

Thus, for example for the source annotation $\tau(G_{\text{want}})$ in Figure 5.13(a), we have $L(\mathsf{O}) = \mathsf{O}[\mathsf{S} \to \mathsf{S}]$ or for short $\mathsf{O}[\mathsf{S}]$. For $\tau(G_{\text{persuade}})$ in Figure 5.13(b), we have

$L(\mathsf{O2}) = \mathsf{O2}[\mathsf{S} \to \mathsf{O}]$. Note that this is also how we write the annotations in the constants G_{want} and G_{persuade} in Figures 5.10(c) and 5.11(c).

For a more complex example, let us turn to the graph in Figure 5.14(a). Here, $L(\mathsf{B}) = \mathsf{B}[\mathsf{C}, \mathsf{E} \to \mathsf{D}]$. We could then write $L(\mathsf{A})$ in long form as

$$L(\mathsf{A}) = \mathsf{A}[\mathsf{A} \to \mathsf{B}[\mathsf{C} \to \mathsf{C}, \mathsf{E} \to \mathsf{D}], \mathsf{C} \to \mathsf{C}, \mathsf{D} \to \mathsf{D}].$$

However, we can shorten $\mathsf{C} \to \mathsf{C}$ to C and the same for D. Further, since C and D are already mentioned in $L(B)$, we can leave them out of the top level of $L(\mathsf{A})$, leaving us with

$$L(\mathsf{A}) = \mathsf{A}[\mathsf{A} \to \mathsf{B}[\mathsf{C}, \mathsf{E} \to \mathsf{D}]].$$

For a full source annotation τ, we then use the string

$$[L(\alpha_1), \ldots, L(\alpha_k)]$$

where $\alpha_1, \ldots \alpha_k$ are the origins in τ. For example, we write $\tau(G_{\text{want}})$ as $[\mathsf{O}[\mathsf{S}]]$ and $\tau(G_{\text{persuade}})$ as $[\mathsf{S}, \mathsf{O2}[\mathsf{S} \to \mathsf{O}]]$. For the source annotation in Figure 5.14(a), we write

$$[\mathsf{A}[\mathsf{A} \to \mathsf{B}[\mathsf{C}, \mathsf{E} \to \mathsf{D}]], \mathsf{E}].$$

We can now formally define annotated s-graphs.

Definition 5.3 (As-graph)**.** An *annotated s-graph* or *as-graph* is a pair $G = \left(\hat{G}, \tau(G)\right)$ of an s-graph $\hat{G} = \left(\hat{G}^\circ, \phi_{\hat{G}}\right)$ and a source annotation $\tau(G)$ such that

(i) G has a root source, i.e. $\mathsf{R} \in \mathit{Src}\left(\hat{G}\right)$ and

(ii) all non-root sources $\alpha \in \mathit{Src}\left(\hat{G}\right) \setminus \{\mathsf{R}\}$ are nodes in $\tau(G)$, and

(iii) every node in $\tau(G)$ is dominated by a source in $\mathit{Src}\left(\hat{G}\right)$.

To simplify notation, we write G° for $\hat{G}^\circ$, ϕ_G for $\phi_{\hat{G}}$, V_G for $V_{\hat{G}}$, and so on.

The idea is that the annotation for a source $\alpha \in \mathit{Src}(G)$ is the subgraph dominated by α in $\tau(G)$. We write $\tau(G)(\alpha)$ for this subgraph. If for example $\tau(G)$ is the graph in Figure 5.14(a), then the annotation at source B is the DAG in (b) of the same figure. In the annotation $\tau(G_{\text{want}})$ in Figure 5.13(a), the annotation for O is the whole graph $\tau(G_{\text{want}})$ including the edge to S, representing the requirement that the O argument have an S source. By contrast, the annotation for S in $\tau(G_{\text{want}})$ is only the node S itself, indicating that no sources are required in the S argument of G_{want}.

Condition (ii) guarantees that such an annotation $\tau(G)(\alpha)$ exists for every source α in the graph, and Condition (iii) ensures that the source annotation $\tau(G)$ is minimal, i.e. only contains information about annotations of sources actually existing in $\hat{G}$. Note that Condition (iii) is equivalent to requiring that every origin of $\tau(G)$ be a source in $\hat{G}$. The root source R plays a special role in the AM algebra, and is thus required in every graph. A graph only containing a root source R, such as G_{James} of Figure 5.9 or G_{raven} of Figure 5.10 has the empty graph as its source annotation.

We also call $\tau(G)$ the *type* of G, and discuss in what sense that turns the AM algebra into a typed algebra in Section 5.3.

We can understand what the annotation $\tau(G)(\alpha)$ on a source α means for the apply operation as follows. The labels on *edges* leaving α describe what sources are *required* to be in the argument, and the *nodes* in $\tau(G)(\alpha)$ describe what these sources will be *unified with* in the apply operation. For example, in $\tau(G_{\text{persuade}})$ in Figure 5.13, the $\mathsf{O2}$ argument is required to have an S source (on the edge label) which will be unified with the O source (edge target). The following definition formalizes the notion of what is required from the argument, adding the condition that the source annotations at the argument also need to match.

Definition 5.4 (Request)**.** Let G be an as-graph and $\alpha \in \mathit{Src}(G)$ one of its sources. Then the *request of G at α* is the DAG $\mathsf{req}_G(\alpha)$ we obtain from $\tau(G)(\alpha)$ by removing α and replacing all other nodes u by the label on the edge (α, u). Formally, we have $\mathsf{req}_G(\alpha) = \left(V_{\mathsf{req}_G(\alpha)}, E_{\mathsf{req}_G(\alpha)}, \lambda_{\mathsf{req}_G(\alpha)}\right)$, where

$$\begin{aligned}
V_{\mathsf{req}_G(\alpha)} &= \left\{\lambda_{\tau(G)}(\alpha, u) \mid u \in V_{\tau(G)(\alpha)} \setminus \{\alpha\}\right\}\\
E_{\mathsf{req}_G(\alpha)} &= \left\{\left(\lambda_{\tau(G)}(\alpha, v), \lambda_{\tau(G)}(\alpha, u)\right) \mid (v, u) \in E_{\tau(G)(\alpha)}, v \neq \alpha\right\}\\
\lambda_{\mathsf{req}_G(\alpha)} &= \left\{\left(\lambda_{\tau(G)}(\alpha, v), \lambda_{\tau(G)}(\alpha, u)\right) \mapsto \lambda_{\tau(G)}(v, u) \mid (v, u) \in E_{\tau(G)(\alpha)}, v \neq \alpha\right\}.
\end{aligned}$$

Let us now move on to the definitions of the apply and modify operations. For a source annotation $\tau(G)(\alpha)$, we define the *annotated renaming function* $R(\tau(G)(\alpha))$ as a permutation on $\mathcal{S}$ that for each edge $e = (\alpha, v)$ in $\tau(G)(\alpha)$ maps the edge label $\lambda_{\tau(G)(\alpha)}(e)$ to the edge target v, and maps R to R.[3]

[3]Note that such a permutation exists since each such edge e has a different label, and no two edges go to the same source v; in other words, we simply need to extend an injective partial function to a permutation. If there are multiple such permutations, we choose an arbitrary one; the choice will not matter since we will only apply this renaming operation to graphs where the sources are among the edge labels and R.

Definition 5.5 (Apply operation APP)**.** Let $G_1 = \left(\hat{G}_1, \tau(G_1)\right)$, $G_2 = \left(\hat{G}_2, \tau(G_2)\right)$ be as-graphs. Then we let $\textsc{app}_\alpha(G_1, G_2) = \left(\hat{G}, \tau(G)\right)$ such that

$$\begin{aligned}\hat{G} &= fg_\alpha\left(\hat{G}_1 \mid\mid ren_{\{\mathsf{R}\leftrightarrow\alpha\}}\left(ren_{R(\tau(G_1)(\alpha))}\left(\hat{G}_2\right)\right)\right)\\ \tau(G) &= \tau(G_1) - \alpha\end{aligned}$$

if and only if

(i) α is an origin in $\tau(G_1)$ (in particular, α is a source in $\hat{G}_1$), and

(ii) the source annotation of G_2 matches the request of G_1 at α, i.e. $\tau(G_2) = \mathsf{req}_{G_1}(\alpha)$.

Otherwise $\textsc{app}_\alpha(G_1, G_2)$ is undefined.

That is, on the s-graph side, we first apply the annotated rename of G_1's α-annotation, i.e. $ren_{R(\tau(G_1)(\alpha))}$, to $\hat{G}_2$, then rename the root R to α with $ren_{\{\mathsf{R}\leftrightarrow\alpha\}}$ and merge.[4] Finally we forget α, i.e. the slot has been filled and α consumed. Accordingly, we remove the α-annotation from $\tau(G_1)$. Condition (i) formulates the basic presupposition that α exists as a source in G_1, also guaranteeing that the resulting graph will be connected (if G_1 and G_2 were connected). Furthermore, Condition (i) makes sure that all 'promises' of a reentrancy (or unification, if you will) given by the source annotation $\tau(G_1)$ will be kept. If α is dominated by another source α' in $\tau(G_1)$, then this states that some source of the future argument of α' will be unified with α, and if we fill α prematurely now, that unification can no longer happen. Looking at the object control example in Figure 5.10 again, if we were to allow $\textsc{app}_\mathsf{S}(G_\text{want}, G_\text{raven})$ first, then inserting G_learn in the O slot later would no longer create a reentrancy since the S source of G_want is already filled, but would instead add a dangling S source; an undesirable result. We will prove a formal statement of how exactly reentrancies are guaranteed in Section 5.2.2.

Condition (ii) reflects the idea that the annotation defines what graphs can be arguments. We can understand the ideas behind Condition (ii) more clearly if we break it down into more fundamental parts.

Lemma 5.6. *Condition (ii) of Definition 5.5 holds if and only if*

[4]Technically, $ren_{\mathsf{R}\leftrightarrow\alpha}$ does not just rename R to α, but swaps the two sources. However, the operation $ren_{R(\tau(G_1)(\alpha))}\left(\hat{G}_2\right)$ renamed all sources in $\hat{G}_2$ to sources in $\tau(G_1)(\alpha)$ that are not α. Thus, in the result of $ren_{R(\tau(G_1)(\alpha))}\left(\hat{G}_2\right)$, there is no α source and $ren_{\mathsf{R}\leftrightarrow\alpha}$ effectively only renames R to α.

(i) all non-root sources in Src (G_2) *are nodes in* $\mathsf{req}_{G_1}(\alpha)$*,*

(ii) for every source α' *in Src* (G_2)*, its annotation* $\tau(G_2)(\alpha')$ *is equal to the subgraph dominated by it in* $\mathsf{req}_{G_1}(\alpha)$*, and*

(iii) any node in $\mathsf{req}_{G_1}(\alpha)$ *is either a source in Src* (G_2)*, or dominated by a source in Src* (G_2) *in* $\tau(G_2)$*.*

Proof. First, let us assume that Condition (ii) of Definition 5.5 holds, i.e. $\tau(G_2) = \mathsf{req}_{G_1}(\alpha)$. Then (i) follows from the fact that all non-root sources of G_2 must be in $\tau(G_2)$ by the definition of an as-graph; (ii) from the definition of $\tau(G_2)(\alpha')$, and (iii) from Condition (iii) of the as-graph definition 5.3.
Conversely, if we assume that (i-iii) hold, we get $\tau(G_2) \subseteq \mathsf{req}_{G_1}(\alpha)$ directly from (i) and (ii), and $\tau(G_2) \supseteq \mathsf{req}_{G_1}(\alpha)$ from (iii). □

Interpreting the conditions of the lemma, (i) and (ii) ensure that G_2 does not bring anything unexpected to the table, and (iii) ensures that G_2 has all sources and annotations G_1 requests. This concludes the definition of apply, and we can move on to modification.

Definition 5.7 (Modify operation MOD)**.** Again, let $G_1 = \left(\hat{G}_1, \tau(G_1)\right)$, $G_2 = \left(\hat{G}_2, \tau(G_2)\right)$ be as-graphs. Then we let $\text{MOD}_\alpha(G_1, G_2) = \left(\hat{G}, \tau(G)\right) = G$ such that

$$\begin{aligned} \hat{G} &= \hat{G}_1 \mid\mid ren_{\{\alpha\leftrightarrow \mathsf{R}\}}\left(fg_{\mathsf{R}}\left(\hat{G}_2\right)\right) \\ \tau(G) &= \tau(G_1) \end{aligned}$$

if and only if

(i) α is an origin in $\tau(G_2)$, in particular G_2 has an α source,

(ii) $\mathsf{req}_{G_2}(\alpha) = (\emptyset, \emptyset, \emptyset)$, i.e. G_2 does not have complex expectations at α, and

(iii) the unified annotations of G_2 without α are a subgraph of the unified annotations of G_1, in other words $\tau(G_2) - \alpha \subseteq \tau(G_1)$.

otherwise $\text{MOD}_\alpha(G_1, G_2)$ is undefined.

The modify operation is slightly simpler than apply, since no complex renames occur. On the graph side, we first forget the root R of $\hat{G}_2$ and then rename α to R.[5] We then merge the graphs, attaching $\hat{G}_2$ to the root of $\hat{G}_1$ where previously

[5]Again, technically $ren_{\{\alpha\leftrightarrow \mathsf{R}\}}$ swaps α and R, but since we just forgot the R source, effectively only α is renamed to R.

$\hat{G}_2$ had its α source. We simply keep the source annotation of G_1, reflecting the intuition that modification does not change the type of the head. Condition (iii) ensures that no information is lost by us only keeping the source annotation of G_1, and that in fact for all sources in G_2, we can still look up an annotation after the modification. At this point, note how the formal definitions indeed match the earlier informal discussion. It remains to show that nothing 'breaks' with these operations.

Lemma 5.8. *Let G_1, G_2 be as-graphs and $\alpha \in \mathcal{S}$. Then, if the operation is defined, $H = \textsc{app}_\alpha(G_1, G_2)$ is a an as-graph. The same holds for $K = \textsc{mod}_\alpha(G_1, G_2)$.*

Proof. To show that $H = \textsc{app}_\alpha(G_1, G_2)$ is an as-graph, we need to show Conditions (i-iii) of Definition 5.3, and that $\tau(H)$ is a source annotation. Clearly, application does not remove the root source R of G_1, so Condition (i) of Definition 5.3 holds.
Now it is crucial to keep in mind that the annotation $\tau(H)$ is the same as $\tau(G_1)$, except with α and the outgoing edges of α removed. At the same time, α is also not a source in H, since the last HR operation that is executed to compute H is fg_α. Thus, all other non-root sources from G_1 are in $\tau(H)$. Furthermore, the rename $R(\tau(G_1)(\alpha))$ applied to G_2 guarantees that all sources coming from the G_2 side are in $\tau(H)$ as well, which means that Condition (ii) of Definition 5.3 holds as well.
Let us now show Condition (iii), that every node in $\tau(H)$ is dominated by a source in $Src(H)$. Since this was the case before for G_1, we only need to consider the descendants of α, since only α gets removed from the annotation. However, all direct descendants of α correspond to origins in the request $\mathsf{req}_{G_1}(\alpha)$, which equals the type of G_2. All the origins of $\mathsf{req}_{G_1}(\alpha)$ must thus be sources of G_2 by Condition (iii) of the as-graph definition 5.3, applied to G_2. Therefore, all sources directly dominated by α in $\tau(G_1)$ come in from the G_2 side in the HR term, and are thus sources in H. This is enough to satisfy Condition (iii) of Definition 5.3.
It is clear to see that $\tau(H)$ is a source annotation, since only the node α gets removed from $\tau(G_1)$, and removing a node cannot interfere with any of the Conditions (i-iv) of Definition 5.2.
The case for $K = \textsc{mod}_\alpha(G_1, G_2)$ is simpler, since here the source annotations do not change, $\tau(K) = \tau(G_1)$. Conditions (i) and (iii) of Definition 5.3 are thus immediately clear, and also that $\tau(K)$ is a source annotation. The Condition (ii) of Definition 5.3, that all non-root sources $\beta \in Src(K)$ are nodes in $\tau(H)$, follows immediately from Condition (iii) in Definition 5.7 of $\textsc{mod}_\alpha$, that $\tau(G_2) - \alpha \subseteq \tau(G_1) = \tau(K)$. That is, all non-root sources of G_2, except α which is renamed to R in the modification process, are in $\tau(H)$. All non-root sources of G_1 are of course in $\tau(G_1) = \tau(K)$. Thus, all non-root sources of K are in $\tau(K)$, which concludes the proof. $\square$

We can now define the AM algebra.

Definition 5.9. Let $\mathcal{S}$ be a finite set of sources and let $\mathfrak{C}$ be a finite set of constant symbols that each denote an as-graph over $\mathcal{S}$. Consider the signature

$$\Sigma = \mathfrak{C} \cup \{\textsc{app}_\alpha, \textsc{mod}_\alpha \mid \alpha \in \mathcal{S}\}\,.$$

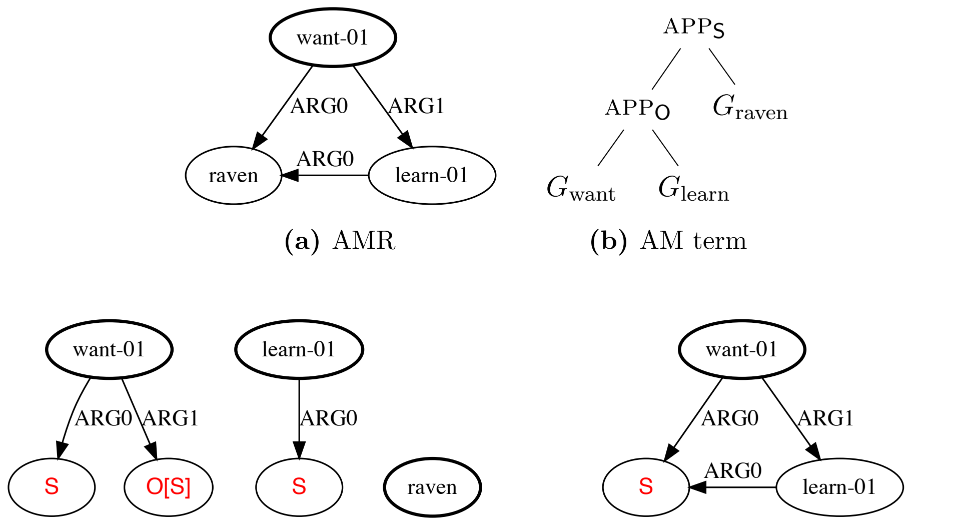
Then the *AM algebra with constants in* $\mathfrak{C}$ *and sources in* $\mathcal{S}$ is the algebra with signature Σ, with domain the set of as-graphs over $\mathcal{S}$, and evaluating the operations as described in this section. The constants in $\mathfrak{C}$ each evaluate to the as-graph they denote.

It is worth emphasizing that the AM algebra is a *partial* algebra, that is, there can be arguments for which the binary operations are not defined. This is because the definitions of $\textsc{app}_\alpha$ and $\textsc{mod}_\alpha$ contain conditions on the arguments, and if these conditions are not met, the operation fails and is undefined. Thus, other than for the HR algebra, not all terms over the AM algebra evaluate to a result. Section 5.3 establishes a type system that allows to easily check whether a term will evaluate or not. But first we can prove a first formal result about the AM algebra, that reentrancies are indeed guaranteed by the annotations in the constants.

5.2.2 Reentrancy guarantees

When discussing e.g. the constant for the control verb *want*, i.e. G_{want} of Figure 5.10, here reprinted in Figure 5.15, we can say that the annotation $\mathsf{O[S]}$ encodes the reentrancy we observe in the resulting AMR. In this section, we will prove that such a reentrancy is in fact guaranteed.

That is, we want to show that for a constant like G_{want}, in the final result of any term it occurs in, there will be a path from the node marked with O to the one marked with S. To achieve this, we need to be able to track these nodes through the term, that is, we need to know which nodes of G_{want} end up where in the final graph. We use a *concrete* version of the AM algebra for this, that is, a version where the domain is the concrete as-graphs, and thus the constants are concrete as-graphs as well (recall the distinction between concrete and abstract graphs of Section 2.2). The $\textsc{app}$ and $\textsc{mod}$ operations stay mostly the same, except that they use the concrete versions of the HR operations inside. Recall that for the concrete merge operation, as described in Section 2.4, nodes that get merged together due to their sources must be identical nodes (as objects) in both graphs in the first place. In particular if the merge operation does not change any node identities, the set of nodes after the



(a) AMR **(b)** AM term

(c) Constants G_{want}, G_{learn} and G_{raven} **(d)** Result of APP$_O$ $(G_{\text{want}}, G_{\text{learn}})$

Figure 5.15: Reprint of Figure 5.10. AMR and its analysis for *The raven wants to learn.*

merge is simply the union of the nodes in both merged graphs. Furthermore, if the nodes don't fulfill this identity condition, the merge can fail. Thus, in the concrete AM algebra, operations can fail even if they are well typed.

As mentioned in earlier chapters, such a concrete algebra is not particularly useful for constructing graphs from scratch – the constants need to know where their nodes will end up in the final result, but this should really instead be defined by the term's structure and operations. However, for a theoretical result such as this, it is still useful. One way to think about this is the following. We can take a term, and evaluate it in the abstract AM algebra as usual. Then we can choose a concrete representative G of the evaluation result. We then trace back through the term which constants created which part of G, and choose concrete constants accordingly. The resulting concrete AM term is then guaranteed to evaluate, and evaluates to G. This allows us to post-hoc associate constants with parts of the evaluation result and keep track of these parts throughout evaluation, even though we started with an abstract AM term. We can now state the reentrancy guarantee as follows.

Proposition 5.10. *Let t be a term over the concrete AM algebra that only uses connected graph constants and that evaluates to an as-graph with empty source annotation (i.e. no non-root sources). Let c be a constant in t, i.e. c is a concrete as-graph, such that there is a node v with source α, a node w with source β and an edge from α to β in the source annotation of c. Then in the evaluation result $[\![t]\!]$, there is a (undirected) path from v to w that does not use any edges of c.*

Proof. First note that the source at v starts out as α, but may be renamed to other sources before being removed. Since the source at v has a non-trivial annotation (it has an edge to the source at w), it cannot be consumed by modification; thus, it can only be renamed to non-root sources. Then it must at some point be removed, since the result of t has no non-root sources. Thus, the source at v must be consumed by an application operation.

To be precise, there must be a position p in t where the left child (call it H_1) has a source (call it α') at v, but the result (call it K) does not have a source at v. To be precise, we define K as the evaluation result of the subterm starting at p, and H_1 as the result of the subterm starting at $p1$. These are concrete as-graphs; also note that the graph c must be a subgraph of H_1. Additionally, we call the right child H_2, this is the result of the subterm starting at $p2$.

H_1 also has a source β' at w; because the source at w existed in $c \subseteq H_1$ and cannot have been removed yet since there is an edge from the source at v to the source at w, thus the source at w can't have been an origin in the source annotation yet, and only origins can get removed.

Further, the source annotation for α' in H_1 contains β', and thus the request of α' in H_1, $\mathsf{req}_{H_1}(\alpha')$, has a source γ that will be renamed to β' during application. In fact, γ is an origin in $\mathsf{req}_{H_1}(\alpha')$ since there is a direct edge from α' to β' in $\tau(H_1)$. Thus, γ must be a source in H_2, and since it will be unified with β', and we work with concrete s-graphs here, γ must also be on the node w. Furthermore, the root (i.e. the R source) of H_2 will be unified with α' in H_1 during application, which means that the root node of H_2 must be v. Since H_2 is connected (all the constants start out connected, and no HR operations disconnect graphs), there must be a (undirected) path P in H_2 from v to w. For the concrete merge operation, the edges of the children must be disjoint (c.f. 2.11), and thus the edges of this path P cannot be in H_1 , and thus also not in c. However, P will exist in $[\![t]\!]$, which concludes the proof. □

5.3 Types

In this section, we show how we can check whether an AM term evaluates without actually running through the whole process, by just looking at the types.

Definition 5.11. Let $G = \left(\hat{G}, \tau(G)\right)$ be an as-graph. We define its *type* to be its source annotation $\tau(G)$.

We can define the apply and modify operations on types only, with slight modifications. We add a special type FAIL to our domain, as a result for operations that are undefined in the AM algebra. Note that Definition 5.4 only depends on the graph's type, and we can therefore extend the definition naturally to all source annotations.

Definition 5.12. Let τ_1, τ_2 be source annotations or FAIL, and α a source. Then we define the result of $\text{APP}_\alpha(\tau_1, \tau_2)$ to be the source annotation $\tau_1 - \alpha$ if

(i) τ_1, τ_2 are source annotations, i.e. $\tau_1, \tau_2 \neq \mathsf{FAIL}$.

(ii) α is a root in τ_1, and

(iii) τ_2 matches the request of τ_1 at α, i.e. $\tau_2 = \mathsf{req}_{\tau_1}(\alpha)$.

Otherwise, the result is FAIL.

Definition 5.13 (Modify operation (MOD))**.** Let τ_1, τ_2 be source annotations or FAIL, and α a source. Then we define the result of $\text{MOD}_\alpha(\tau_1, \tau_2)$ to be τ_1 if

(i) α is a root in τ_2,

(ii) $\mathsf{req}_{\tau_2}(\alpha) = (\emptyset, \emptyset, \emptyset)$, i.e. τ_2 does not have complex expectations at α, and

(iii) τ_2 without α is a subgraph of τ_1.

Otherwise the result is FAIL.

Definition 5.14. We define the *algebra of AM types* to have as domain the source annotations over some source set $\mathcal{S}$ plus the special object FAIL. The algebra's signature is $\Sigma_{\mathfrak{C}}$ of Definition 5.9. The as-graph constants evaluate to the graph's type, and the other operations evaluate as just described.
We say a term is *well-typed* if it evaluates to a type different from FAIL in the algebra of AM types.

Theorem 5.15. *A term t evaluates to an as-graph over the AM algebra if and only if it is well-typed.*

Proof. This follows inductively (over the depth of t) from the fact that the conditions for the operations on types precisely mirror the conditions for the operations on AM graphs, and same for the operation results. □

5.4 Discussion

At the start of this chapter, we set out to reduce both functional and surface ambiguities of the HR algebra, by designing an algebra that is capable of exactly application, modification and unification. And indeed, the AM algebra supports these mechanisms, and since the merge, rename and forget operations of the HR algebra all have a fixed place inside the apply and modify operations of the AM algebra, most causes of surface ambiguities should be addressed. But how much ambiguity remains? Or did we maybe restrict the operations too much, such that many graphs might have no derivation anymore? And are the derivations we get consistent with the mechanisms we saw for the compositional formalisms in Section 3.2? In this section, we look at several examples to answer these questions, and give a qualitative examination of the AM algebra. An empirical, quantitative evaluation follows in Section 5.6.

We will discuss the examples of Section 3.2, and the reader is invited to recall that section. Some of these examples are interesting mostly because of their relation to the syntax of the sentence. Since we only talk about semantic terms here – we will discuss how we relate AM terms to sentences in the next chapter – we will not go into detail about the syntax heavy examples here; we will do that in the next chapter instead.

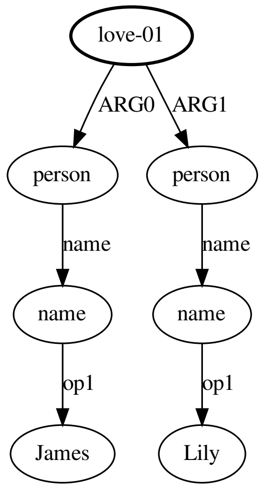
5.4.1 Examples

Figure 5.16 shows examples of simple application (a-c) and modification (d-f), just as we saw earlier. At each operation in the terms (b) and (e), the type of the local result is written in red. For the application example, there are just two terms, the one in (b) and one with the order of applications swapped. For the modification example, the term in (e) is the only one describing the AMR with these constants. Thus, for these two graphs, ambiguity is drastically reduced. At the same time, the available terms are very close to how the IRTG of Koller (2015) handles these phenomena, i.e. these are desirable terms. We discuss how we obtain constants like this in practice in Section 5.5.

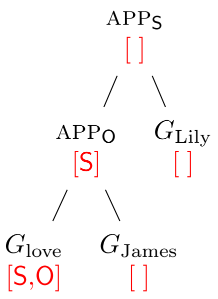
Control and raising. Recall now the subject control example of earlier, namely the sentence

(1) The raven wants to learn.

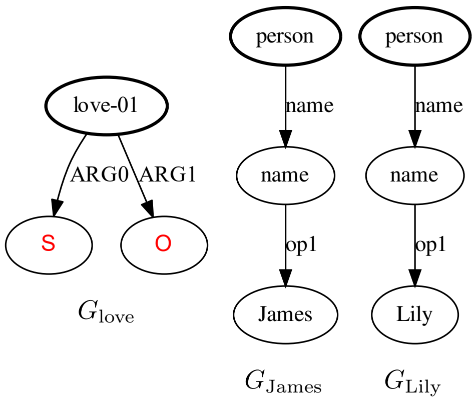
here printed in Figure 5.17. The AMR in (a) is created by the term (b) – again, the only term given these constants –, where the subject source S of G_{learn} is expected by



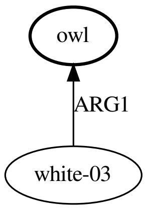
(a) AMR for *James loves Lily.*



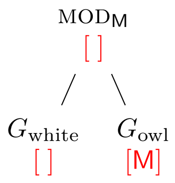
(b) AM term for the graph in (a).



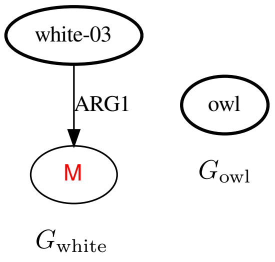
(c) Constants used in (b).



(d) AMR for *a white owl.*

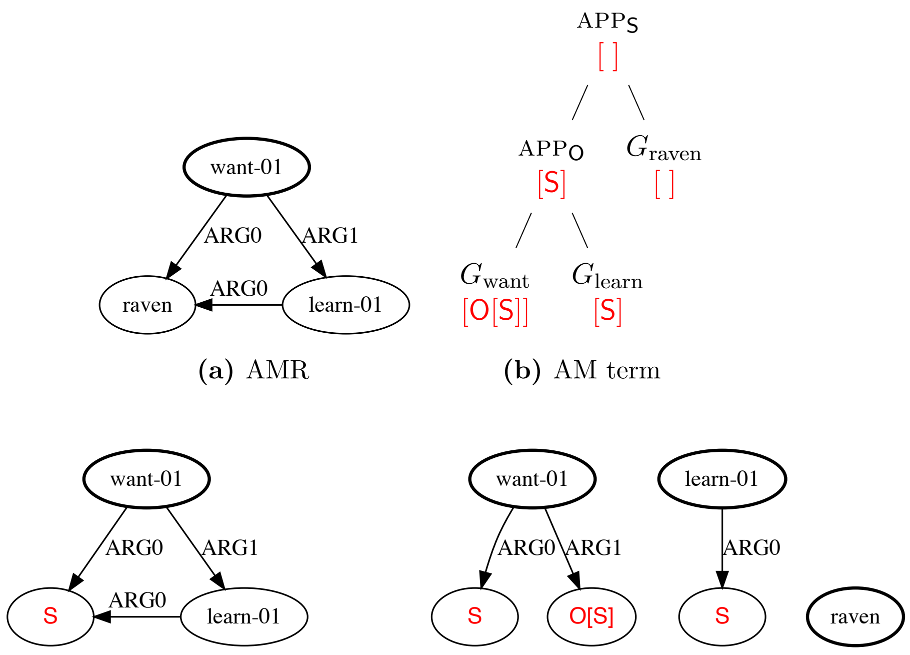


(e) AM term for the graph in (b).



(f) Constants used in (e).

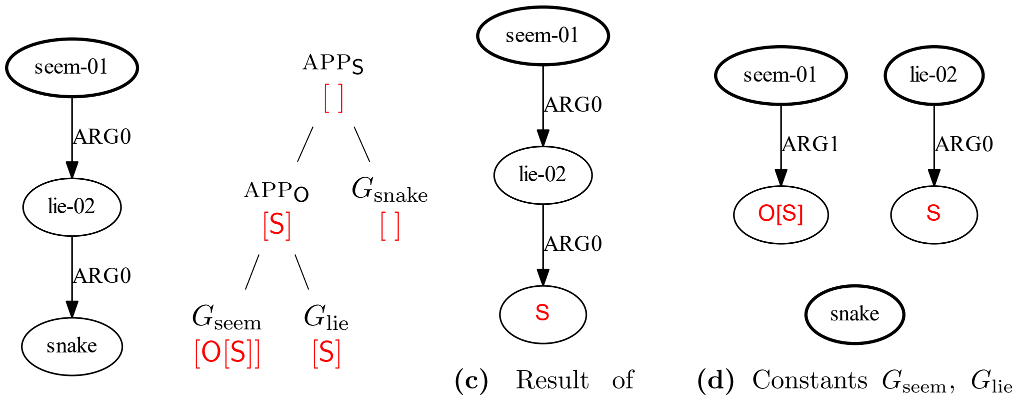
Figure 5.16: Example analyses of (a-c) *The raven learns* and (d-f) *dangerous spell*, illustrating application and modification.



(a) AMR **(b)** AM term

(c) Result of APP$_O$ $(G_{\text{want}}, G_{\text{learn}})$ **(d)** Constants G_{want}, G_{learn} and G_{raven}

Figure 5.17: Subject control in *The raven wants to learn.*



(a) AMR **(b)** AM term **(c)** Result of APP$_O$ $(G_{\text{seem}}, G_{\text{lie}})$ **(d)** Constants G_{seem}, G_{lie} and G_{snake}

Figure 5.18: Raising in *The snake seems to be lying.*

G_{want} (see the constants in (d)). The two subjects get automatically unified to give the as-graph in (c), and G_{raven} is fit into that common subject slot with the $\textsc{app}_{\mathsf{S}}$ operation at the top in (b). In particular, this reentrancy is encoded in the constant for the control verb *want*, as is typical for the compositional formalisms we saw in 3.2. The reader is invited to also recall the object control example in Figure 5.11 – the same observations hold there.

It is interesting to compare the control phenomenon to *raising*. Figure 5.18(a) shows an AMR for the sentence

(2) The snake seems to be lying.

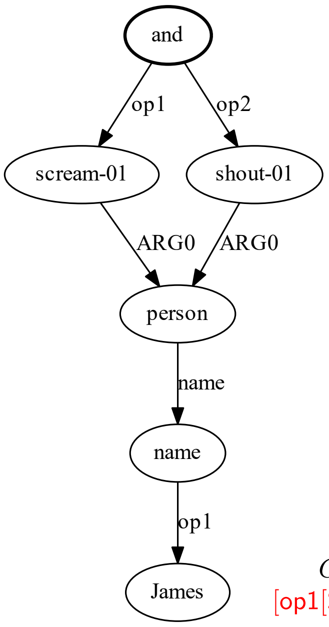
On the surface, this sentence looks a lot like the control case, but the AMR is different: semantically, the snake is not a direct argument of the raising verb *seem-01*. However, we can construct an AM term, namely the one in Figure 5.18(b), that has the same structure as in the control case (Figure 5.17(b)). In this derivation, G_{seem} combines with G_{lie} first to give the result in Figure 5.18(c). Only then is the subject added. This makes the construction of the AMR in Figure 5.18(a) consistent with both the standard syntax of the sentence, and the derivation in the control case. In fact, the graphs G_{want} and G_{seem} have the same type, despite G_{seem} not having an overt S source.

Coordination. The constant $G_{\text{and:}[\mathsf{S}]}$ in Figure 5.19(e) coordinates two graphs of type $[\mathsf{S}]$, i.e. two verbs still missing their subjects. This creates an AMR such as the one in Figure 5.19(a) for *James screams and shouts*, with term in (b). Here, first G_{scream} is added into the $\mathsf{op1}$ slot, yielding the graph in (c). Then, G_{shout} is added as $\mathsf{op2}$ and the S sources of the two verbs merge, see (d). The subject G_{James} is then added as usual. The order in which G_{scream} and G_{shout} are added is arbitrary, such that there are two terms possible here. Coordination for any other type works similarly.[6]

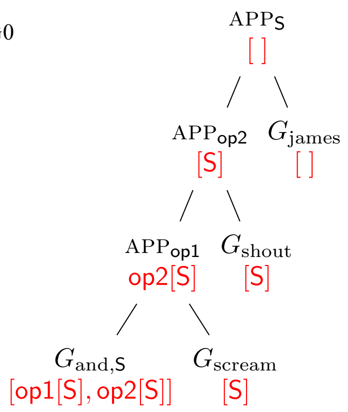
Again, this mechanism mirrors the coordination of like types we saw in compositional formalisms, such as in CCG. While technically, we could use constants with node label *and* (or any other coordination) that expect different types at their opx arguments, in practice, we will only allow constants that coordinate like types (c.f. Section 5.5).

Relative clauses. Recall that relative clauses are different in AMR only through placement of the root source R. For example, compare Figure 5.20(a) for

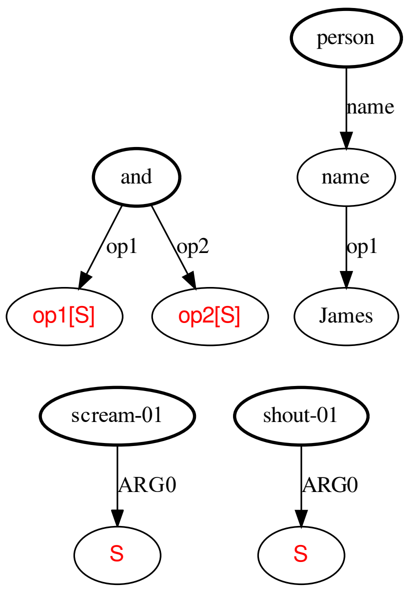
[6]as long as the coordinated type does not contain opx sources, which would accidentally merge with the opx sources of the coordinating graph.



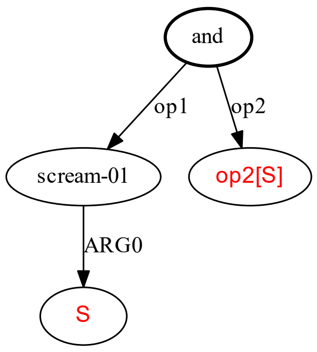
(a) AMR



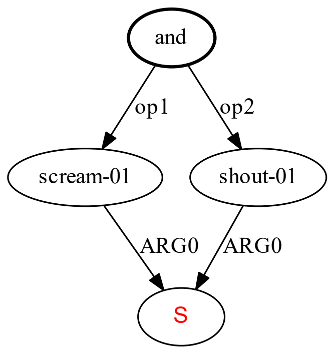
(b) AM term



(c) Constants $G_{\text{and,S}}$, G_{james}, G_{scream} and G_{shout}

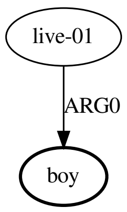


(d) Result of APP$_{op1}$ $(G_{\text{and,S}}, G_{\text{scream}})$

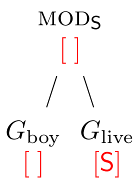


(e) Result of APP$_{op2}$ (APP$_{op1}$ $(G_{\text{and,S}}, G_{\text{scream}})$, $G_{\text{shout}})$

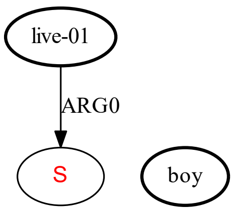
Figure 5.19: Verb coordination in *James screams and shouts.*



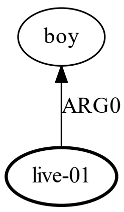
(a) AMR



(b) AM term

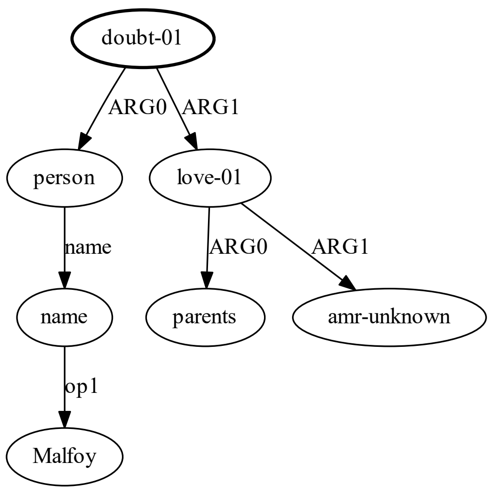


(c) Constants G_{live}, G_{boy}

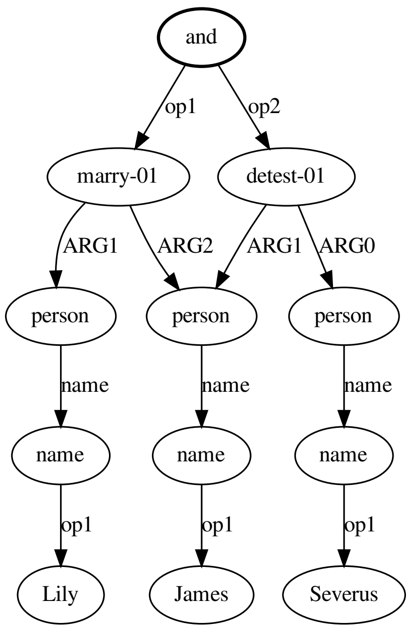


(d) AMR for *The boy lives.*

Figure 5.20: (a-c) A relative clause in *the boy who lives*, and (d) the AMR of *The boy lives* for contrast.



(a) *Who does Malfoy doubt the parents love?*



(b) *Lily married and Severus detests James*

Figure 5.21: AMRs for wh-movement and right node raising.

(3) The boy who lives

and (d) for

(4) The boy lives.

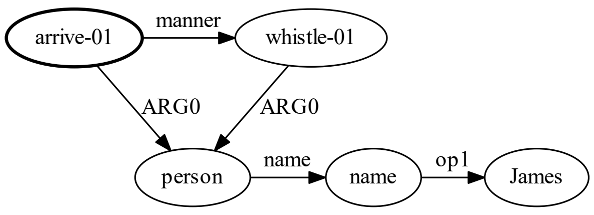
The AM algebra handles relative clauses simply through using modification instead of application on the subject (or object) slot, as shown in the term in Figure 5.20(b).

Wh-movement and non-constituent coordination. These phenomena are mostly difficult because of properties of the syntax. For example, in the long-distance wh-movement in

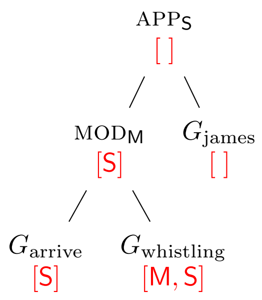
(5) Who$_i$ does Malfoy doubt the parents love $\_\__i$?

the distance between the verb *love* and its object *who* is big, since *who* has moved to the start of the sentence (the usual object position is indicated with $\_\__i$). However, in the corresponding AMR in Figure 5.21(a), the distance between the *love-01* node and the *amr-unknown* node representing *who* is minimal, and the AM algebra has a term for this AMR with just basic application operations.

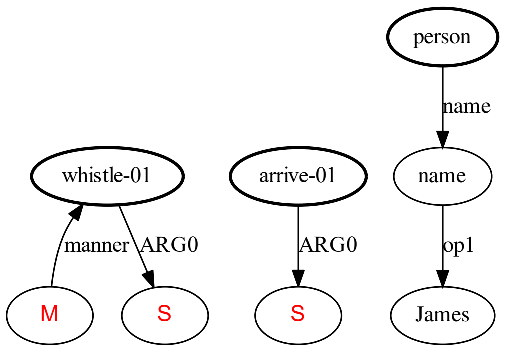
Similarly, non-constituent coordination such as the right node raising in



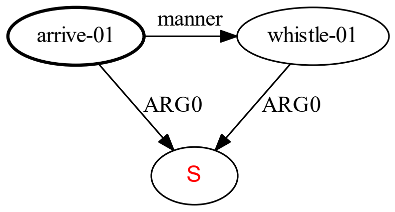
(a) AMR



(b) AM term



(c) Constants $G_{\text{whistling}}$, G_{arrive} and G_{james}



(d) '*arrives whistling*': result of MOD$_{\mathsf{M}}$ $(G_{\text{arrive}}, G_{\text{whistling}})$

Figure 5.22: Secondary predication in the sentence *James arrives whistling*.

(6) Lily married and Severus detests James

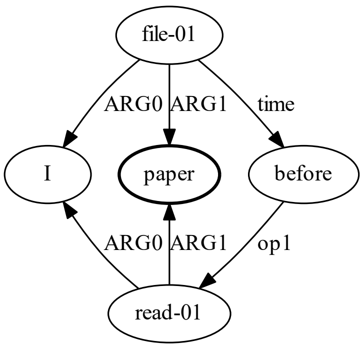
is only a problem with respect to syntax (The phrases *Lily married* and *Severus detests* that get coordinated are not usually considered constituents). The graph, shown in Figure 5.21(b) can be built similarly to the subject coordination above (here, the subject applications occur first for the verbs separately, and then the O sources are coordinated).

We will discuss how our parser can produce these graphs from a sentence in the next chapter.

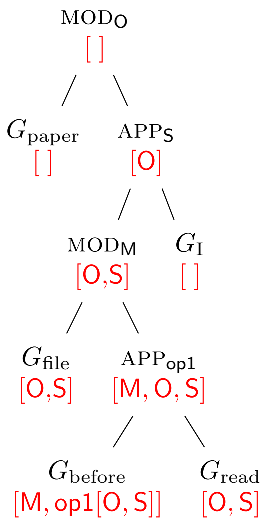
Secondary predication and parasitic gaps. In (7), *whistling* is a secondary predicate that shares its subject with the main verb *arrives*.

(7) James arrives whistling

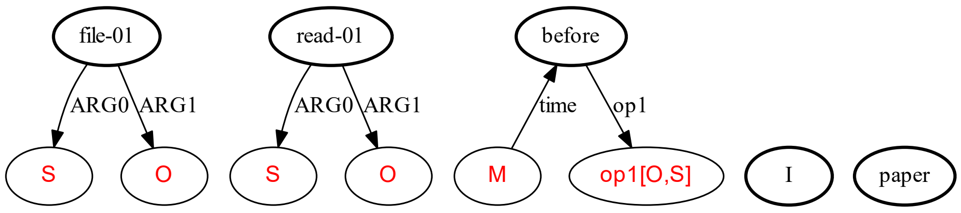
The corresponding AMR is shown in Figure 5.22(a). The AM algebra can build this AMR with modification, see Figure 5.22(b). At the MOD$_{\mathsf{M}}$ operation, the modifier $G_{\text{whistling}}$ still has its S source, which merges with the S source of G_{arrive}, yielding



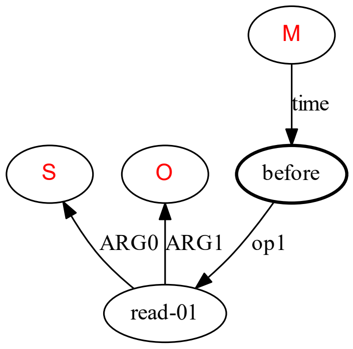
(a) AMR



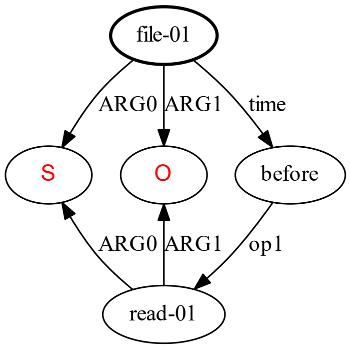
(b) AM term



(c) Constants G_{file}, G_{read}, G_{before}, G_{I} and G_{paper}



(d) '*before reading*': result of $\text{APP}_{\text{op1}}(G_{\text{before}}, G_{\text{read}})$



(e) '*filed before reading*': result of $\text{MOD}_{\text{M}}(G_{\text{file}}, \text{APP}_{\text{op1}}(G_{\text{before}}, G_{\text{read}}))$

Figure 5.23: A parasitic gap in *the paper I filed before reading.*

the graph in (d). Then, $\textsc{app}_{\mathsf{S}}$ fills this slot as usual.
An extension of this is parasitic gaps as in

(8) The paper I filed before reading

where the secondary predicate *reading* not only shares its subject with the main verb *filed*, but also the object; see the AMR in Figure 5.23(a). In the term in (b), first G_{before} is combined with G_{read} via the $\textsc{app}_{\mathsf{op1}}$ operation, passing along the S and O sources of G_{read} to yield the graph in (d). This construct then modifies G_{file} as in the secondary predication in (7) above, now unifying both the S and O sources respectively. Recall that this modification is in fact only allowed because G_{file} has an S and O source itself. The joint S and O slots are then filled as usual.

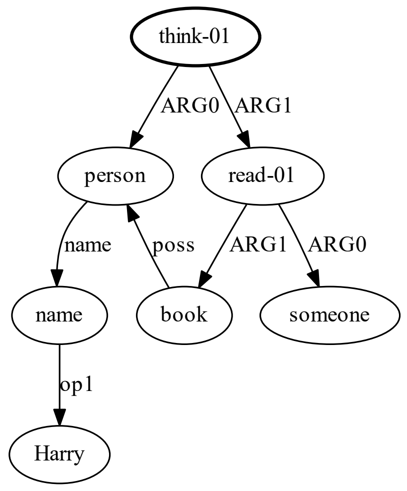
Coreference. This is a phenomenon that the AM algebra cannot properly handle. Take for example the sentence

(9) Harry$_i$ thinks someone is reading his$_i$ books.

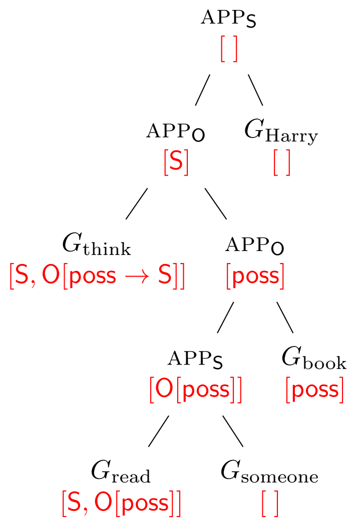
where the i in the index indicates the coreference; with the AMR in Figure 5.24(a). For these constants in (c), G_{book} has a poss slot[7], which somehow needs to be unified with the subject slot of G_{think} before that is filled with G_{Harry}. There is a technical solution shown in (b), where the poss slot is passed through G_{read} via the $\mathsf{O}[\mathsf{poss}]$ annotation, c.f. (d), and then unified in G_{think} via the $\mathsf{O}[\mathsf{poss} \to \mathsf{S}]$ annotation. However, this solution puts the responsibility of resolving the coreference with the constants G_{think} and G_{read}, which is not good: to be properly reusable, these constants should be the same (i.e. without special annotations on their O sources) regardless of whether there is coreference in the rest of the sentence.
There may be solutions to this using e.g. a separate indexing mechanism such as the Skolem IDs used in Artzi et al. (2015). However, this thesis will focus on phenomena that can be handled more straightforwardly in a compositional framework, and simply delete coreferent edges before decomposition when necessary. This is described in more detail in Section 5.5.3.

Completing meaningful subgraphs. In the problem statement at the start of this chapter, we looked at the AMR in Figure 5.25(a), illustrating the problem that the HR algebra does not have to complete meaningful subgraphs before it combines them further. This is different for the AM algebra. Since application arguments may not have any sources besides what the head requests, and their roots R get removed

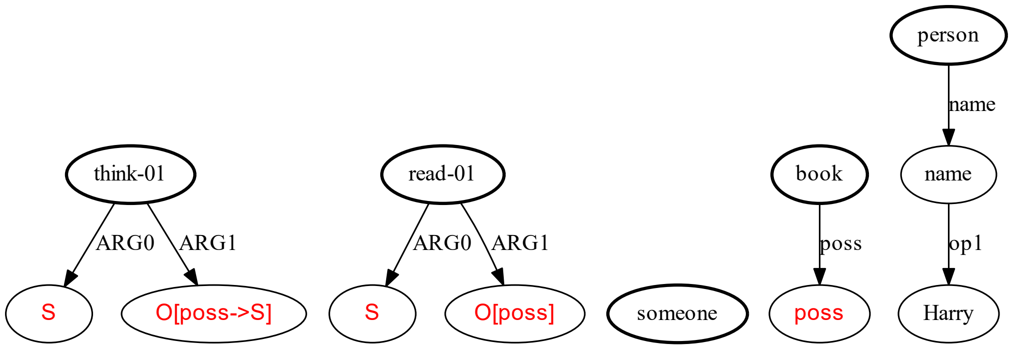
[7]One might also consider introducing the *book* node and the poss edge in separate constants.



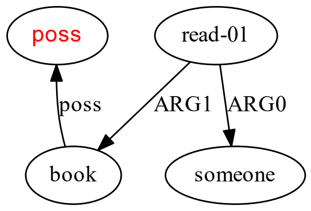
(a) AMR


APP_S [ ]
APP_O [S]
G_Harry [ ]
G_think [S, O[poss → S]]
APP_O [poss]
APP_S [O[poss]]
G_book [poss]
G_read [S, O[poss]]
G_someone [ ]

(b) AM term

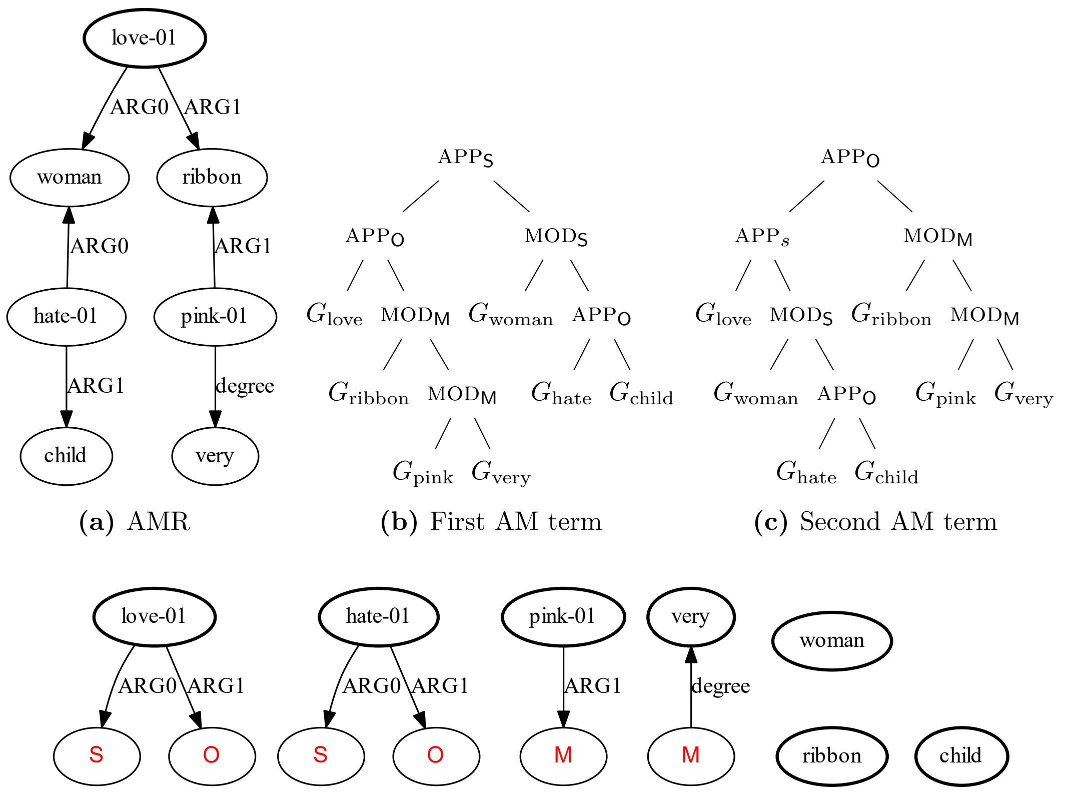


(c) Constants G_think, G_read, G_someone, G_book and G_Harry



(d) '*someone is reading his book*': result of $\text{APP}_\text{O}\left(\text{APP}_\text{S}\left(G_\text{read}, G_\text{someone}\right), G_\text{book}\right)$

Figure 5.24: (a) An AMR for *Harry thinks someone is reading his books.* The analysis in (b-d) is undesirable.



(a) AMR **(b)** First AM term **(c)** Second AM term

(d) Constants G_{love}, G_{hate}, G_{pink}, G_{very}, G_{woman}, G_{ribbon} and G_{child}

Figure 5.25: (a) AMR for *The woman who hates children loves very pink ribbons*, and (b-c) the only two AM terms describing it given the constants in (d).

in the process such that nothing can attach there later, arguments must be complete before application. Here "complete" means that it should not contain any sources besides the ones the head specifically asks for in its annotation. For the AMR in Figure 5.25(a), using the constants in (d), there are thus just the two terms in (b) and (c), differing only in whether the object or subject of G_{love} is added first.

5.4.2 Conclusion

For most of the discussed phenomena, we found analyses that are consistent with the methods used in other compositional formalisms as discussed in Section 3.2. Furthermore, the analyses are consistent *with each other*, for example the constants for intransitive verbs have the same shape in simple sentences (G_{learn} in Figure 5.16), in control sentences (G_{learn} in Figure 5.17) and in coordination (G_{scream} and G_{shout} in Figure 5.19).

Furthermore, surface ambiguities and undesirable functional ambiguities have been reduced drastically, often only allowing one or two terms for these small examples where with the HR algebra there were billions of billions of terms. The apply and modify operations serve as normal forms of HR operations to reduce surface ambiguities, and reduce functional ambiguities with the help of the type system. This allows us (as demonstrated below in Section 5.6) to use more source names, and in particular to use meaningful source names across the board, as we had originally set out to do (how we find those source names in practice is discussed in Section 5.5.1). We are also back to using the larger, more meaningful constants that were typical of the compositional formalisms in Section 3.2.

5.5 Decomposing AMRs with the AM algebra

At this point, we have seen that the AM algebra is built on simple intuitions, but also has a formal foundation that allows supporting the intuitions with technical proofs, such as the reentrancy guarantee in Section 5.2.2. Further, we showed that a range of non-trivial phenomena are analyzed satisfactorily (with some limitations leaving room for growth). With the theory covered, we now tackle the task of computing decomposition automata for the AM algebra in practice.

The decomposition automata follow similar principles as the HR decomposition automata in Chapter 4; in fact, we will re-use much of that method for the s-graph part of as-graphs. But we deal with one new challenge first: when decomposing with the HR algebra, the constants were simple, just single loops and edges that we could read off of the graph. Here, by contrast, we need more complex constants,

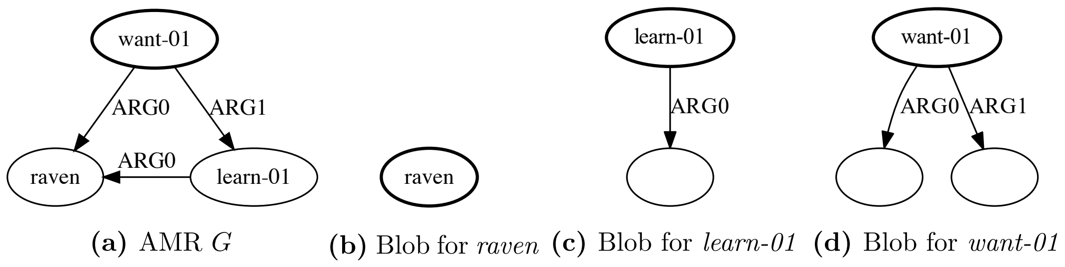


Figure 5.26: (a) An AMR G for *The raven wants to learn*; (b)-(d) the blobs we obtain

with edges attached to nodes already, and we aim for meaningful source names with annotations. We need to obtain all of this automatically from the complete AMR. In the following, we describe heuristic methods to extract candidate constants from a given AMR G. Throughout, we will use a fixed set of sources

$$\begin{aligned}\mathcal{S} = &\{\mathsf{R}, \mathsf{S}, \mathsf{O}, \mathsf{O2}, \ldots, \mathsf{O9}, \mathsf{mod}, \mathsf{poss}, \mathsf{domain}\}\\ &\cup \{\mathsf{opx} \mid x \in \mathbb{N}, \text{op}_x \text{ edge label occurs in the corpus}\}\,.\end{aligned}$$

We will often treat G as a concrete graph in this section, so that we can refer to specific nodes and edges conveniently.

5.5.1 Constants and their types

We start by cutting G up into the subgraphs that will serve as graph backbones of the constants. We use the same graph fragments as we used for the larger constants in Section 4.6, and we refer to these fragments here as *blobs*. To recall, a blob consists of a main labeled node and the node's outgoing edges with an ARGx, opx, sntx ($x \in \mathbb{N}$), domain, poss or part label, and its incoming edges with any other label. We call these edges the *blob edges*. Blobs defined in this way uniquely partition an AMR's edge set. Take the graph in Figure 5.26(a), the blobs are shown in (b-d) with the ARGx edges attached to the predicates they belong to. Peng et al. (2015) use a very similar heuristic to segment a graph into pieces.

Note that the unlabeled endpoints of the blob edges are included in the blobs. We call these unlabeled endpoints of the blob edges the *blob-targets*. We will construct a set of constants for each blob, such that the value of each constant is an as-graph whose graph component is the blob. The main node of the blob will be the R-source. It remains to assign source names and annotations to the blob-targets. The different choices of annotated source names then constitute the different constants for this blob.

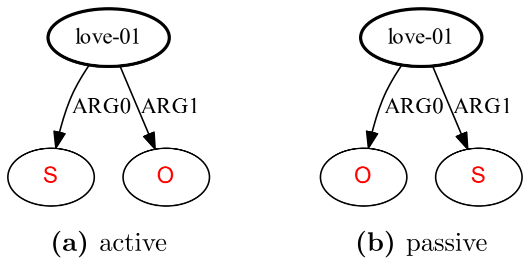
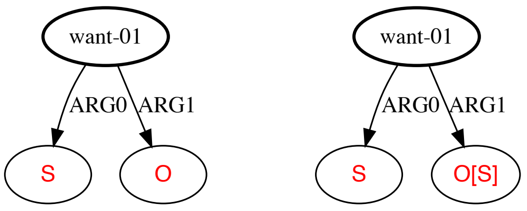


Figure 5.27: Active and passive versions of the constant for a transitive verb.

Source names

We heuristically assign (syntactic) source names from $\mathcal{S}$ to the blob-target nodes based on the edge label of their adjacent edge in the blob. Let v be a node in our graph G, and let us now determine the source names used in the blob of v. Canonically, we use the following edge-to-source mapping E2S to determine sources for v's blob-targets: For most nodes v, E2S maps ARG0 to S; ARG1 to O and other ARG$_x$ to O[x]; poss and part to poss; snt$_x$, op$_x$ and domain to themselves; and all other edges to mod. Exceptionally, if v has a node label that is a conjunction[8] and at least two outgoing ARG$_x$ edges, we map ARG$_x$ to opx instead. E2S determines the canonical *target-to-source mapping* b_v, which assigns a source to each blob-target u: if the edge between v and u has label e, then we have $b_v(u) = \text{E2S}(e)$. For example, for the *want-01* node in Figure 5.26, this mapping gives us the constant in Figure 5.28(a).

A given blob may generate more than one constant, each with different sources on different nodes; accordingly, for each node v in G, we collect a *set* $B(v)$ of such target-to-source mappings. $B(v)$ contains the canonical mapping b_v, and we generate further target-to-source mappings by applying a fixed set of lexical rules to b_v. The *passive* rule switches S with any O, and *object promotion* maps Oi to O(i-1) (let O0=O). We allow all results of such mappings with at most one use of *passive*, that have no duplicate source names. For example, the constant in Figure 5.27(b) is a result of the passive rule; Figure 5.27(a) shows the canonical mapping b_v for v the *love-01* node. Having this flexibility in the source names will help when we handle coordination below. For each mapping in $B(v)$, we create a constant with the respective sources and no further source annotations.



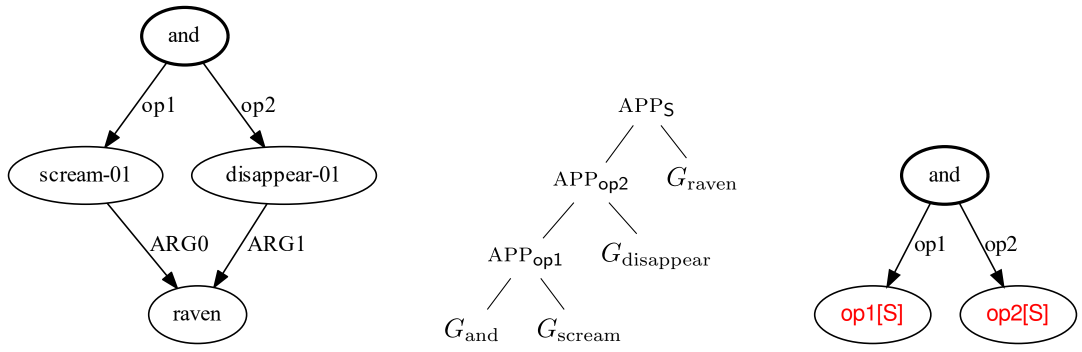
(a) trivial annotations **(b)** non-trivial annotations

Figure 5.28: Possible source annotations for the *want-01* constant for the graph in Figure 5.26(a).

Annotations

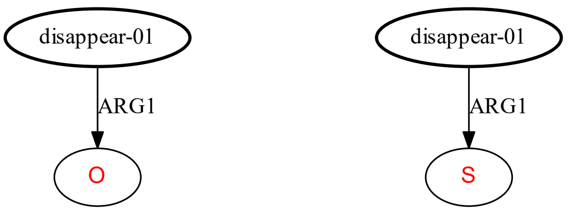
We can also use these target-to-source mappings to extract constants that have sources with non-trivial argument types and renaming functions. Consider the subject-control AMR in Figure 5.26(a) above. So far, we obtain the constant in Figure 5.28(a), but we also want to generate the constant G_{want} in Figure 5.28(b); i.e. determine the edge from O to S in the source annotation $\mathsf{O[S]}$. Writing v_{want}, v_{raven}, and v_{learn} for the *want-01*, *raven*, and *learn-01* nodes of the graph in 5.26(a), note that it is the ARG0 edge from v_{learn} to v_{raven} that signals the control structure. That is, v_{want} has a blob-target v_{learn}, and the two share a *common* blob-target v_{raven}. For such a triangle structure, we consider any target-to-source mappings $m_w \in B(v_{want})$ and $m_l \in B(v_{learn})$. We then add a constant for v_{want} which as before uses the source names of m_w, but now in the source annotation of the *want* constant there is an edge from $m_w(v_{learn})$ to $m_w(v_{raven})$ with edge label $m_l\,(v_{raven})$. This anticipates the open source $m_l\,(v_{raven})$ coming from the v_{learn} constant and ensures unification with the source $m_w(v_{raven})$, renaming from $m_l\,(v_{raven})$ to $m_w(v_{raven})$ if necessary. In other words, we set up the annotation in the v_{want} constant such that when we apply it to a v_{learn} constant that has sources according to m_l, we obtain the structure we found in the graph. Let us consider the canonical example here, with $m_w = \{v_{learn} \mapsto \mathsf{O}, v_{raven} \mapsto \mathsf{S}\}$ and $m_l = \{v_{raven} \mapsto \mathsf{S}\}$. In this case, $m_w(v_{raven}) = m_r(v_{raven}) = \mathsf{S}$, therefore no rename is necessary and we obtain the constant of Figure 5.28(b). If we choose $m_l = \{v_{raven} \mapsto \mathsf{O}\}$ instead, we obtain a constant for the v_{want} blob where the O source is annotated $\mathsf{O[O \to S]}$. In this graph, this is not particularly meaningful from a linguistic perspective, but in other graphs this principle allows us to generate e.g. the object control structure of persuade. To ensure that we recover the correct constant, we simply add constants for all choices of $m_w \in B(v_{want})$ and $m_l \in B(v_{learn})$.

[8]According to the AMR documentation, these are *and*, *or*, *contrast-01*, *either* and *neither*.



(a) AMR for *The raven screams and disappears.* **(b)** AM term for the AMR in (a) **(c)** Matching constant G_{and}.

Figure 5.29: An example of coordination.



(a) canonical source name **(b)** *passive* source name

Figure 5.30: Possible source names for the *disappear-01* constant for the graph in Figure 5.29(a).

Let us now find the constants for the *and* node in Figure 5.29(a). The AMR corresponds to the sentence *The raven screams and disappears.* The algorithm used here restricts constants to coordination of like types, as we observed as a common pattern in Chapter 3. In the intended AM term, shown in Figure 5.29(b), we first coordinate *scream* and *disappear* before we apply the result to the common argument *raven*. To generate the constant for *and*, we consider maps $m_s \in B(v_{scream})$ and $m_d \in B(v_{dis})$, where v_{scream} and v_{dis} are the nodes labeled *scream-01* and *disappear-01* respectively. The *raven* node v_{raven} is a blob-target of both v_{scream} and v_{dis}. If additionally the target-to-source maps agree, e.g. $m_s(v_{raven}) = m_d(v_{raven}) = \mathsf{S}$, we add a new constant for the *and* blob where both op1 and op2 have an edge to S in the source annotation (using S also as the edge label, since we don't want a rename here). This yields G_{and} as depicted in Figure 5.29(c). Here, the *passive* operation for assigning alternative source names comes in handy. The canonical source assignment would produce an O source for $m_d(v_{raven})$, as in Figure 5.30(a). But with the passive rule, we also allow the constant in Figure 5.30(b), allowing $m_d(v_{raven}) = \mathsf{S}$. This allows us to have like types for both coordinated verbs, and also makes sense linguistically. The *unaccusative* verb *disappear-01* has a semantic object (indicated by the ARG1 edge) that is syntactically a subject. Thus, using the S source here is justified. For the case where $m_s(v_{raven}) = \mathsf{S}$ but $m_d(v_{raven}) = \mathsf{O}$, we do not create a new constant. Again, we take all combinations of choices for m_s and m_r into account.

Similar patterns allow us to find possible raised subjects for raising constructions, and to handle coordination of control verbs. Using these patterns recursively, we can handle nested control, coordination and raising constructions. For example in Figure 5.31, finding the *raven* node as a common target in coordination allows us to generate G_{want} analogously to the previous example based on Figure 5.26(a).

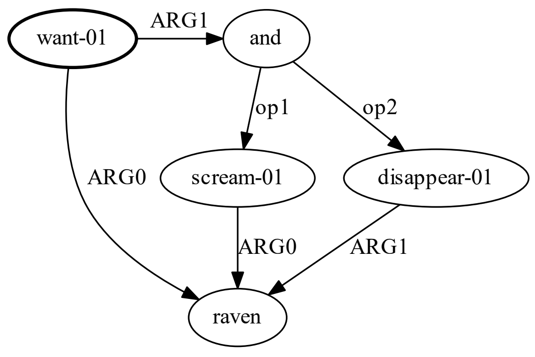


Figure 5.31: AMR for *The raven wants to scream and disappear.*

In sum, we obtain types and renaming functions that cover a variety of phenomena, in particular the ones described in Section 5.4.

5.5.2 Obtaining the set of terms

We define the AM decomposition automaton similarly to the HR case, also using concrete sub-s-graphs to model the behavior of abstract graphs. The correctness of this automaton follows with the same argument as in Chapter 4.

Definition 5.16. Let G be a connected concrete s-graph with only source R. Let Σ_G be the signature described earlier in this section. Then let the *decomposition automaton* $\mathrm{A}_G = (Q, \Sigma_G, Q_f, \Delta)$ where the set of states Q is the set of concrete as-graphs whose s-graphs are sub-s-graphs of G, and the set of final states $Q_f = \{(G, [\,])\}$ contains only G with the empty type. Δ is the set of the following transition rules:

- $\textsc{app}_\alpha(H_1, H_2) \to H_3$ for all $H_1, H_2, H_3 \in Q$ with $H_3 = \textsc{app}_\alpha(H_1, H_2)$.
- $\textsc{mod}_\alpha(H_1, H_2) \to H_3$ for all $H_1, H_2, H_3 \in Q$ with $H_3 = \textsc{mod}_\alpha(H_1, H_2)$.
- $c \to H$ for every $c \in \Sigma_G$ of arity 0, and every sub-s-graph H of G with $H \simeq \hat{c}$.

We can enumerate the rules of this automaton with the standard bottom-up algorithm of Section 2.5. We can check the type constraints on apply and modify straightforwardly, and check the HR rules on the concrete sub-s-graphs in the same way as in Chapter 4.

Source name preferences. In Section 5.5.1, we allowed some flexibility in the source names we assign to blob targets. For example, the *passive* version of a constant has the S and O sources swapped. This allows us e.g. to coordinate verbs in active and passive. However, this flexibility also adds ambiguity to the terms. Using *weighted* tree automata, to describe the terms allows us to add a preference to the source assignments, giving us more consistent terms, while keeping the flexibility. In a weighted tree automaton, each rule has a weight, and the score of a term is the sum of the weights of the rules used in it. We increase the weights of constants that satisfy our preferences. We prefer active over passive, except when there is no ARG0 edge – for example, the presence of an ARG1 edge without an ARG0 edge in a constant often indicates an unaccusative subject, where the verb has a syntactic subject that is a semantic object. For example for the sentence *The lion relaxes*, we would find a constant as in Figure 5.32.

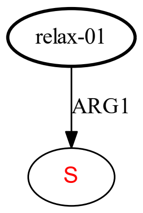


Figure 5.32: We prefer this constant for *relax-01*, using S over O.

There is room for improvement here. For example, one could try to make these decisions on a case by case basis, taking the sentence syntax into account. But this goes beyond this thesis. Here, this source name preference is a simple and practical way to reduce the number of terms without losing coverage (see Section 5.6). In Section 6.8.5 we will see that the source name preferences have a considerable effect on parsing performance.

Lookup. We use a simplified lookup here, using just types and the 'central node' of the operation. That is, for $\textsc{app}_\alpha$, for the left argument G_1 we use $\mathsf{req}_{G_1}(\alpha)$ and the α node of G_1, while for the right argument G_2 we use $\tau(G_2)$ and its root source. Note that these need to match for the application to be successful. For $\textsc{mod}_\alpha$, since the types are not exactly specified here, we only index with the α source of the right argument G_2 and the Rsource of the left argument G_1.

A note on the asymptotic runtime. With a more thorough lookup structure, the asymptotic runtime bounds we established for the HR algebra in Chapter 4 could apply here as well; that is, $O\left(n^s 3^{ds} ds\right)$. However, since we use so many source names here, the bounds would be very high (the number of source names is in the exponent), and in stark contrast to the fast runtimes we achieve in practice (see Section 5.6 below). Combined with the strong type restrictions the AM algebra employs, this hints at the existence of better theoretical runtime bounds for the AM algebra specifically. But at the time of writing, no such improved upper bound is known to the author of this work.

5.5.3 Removing edges

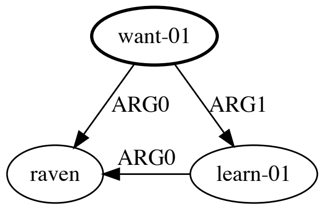
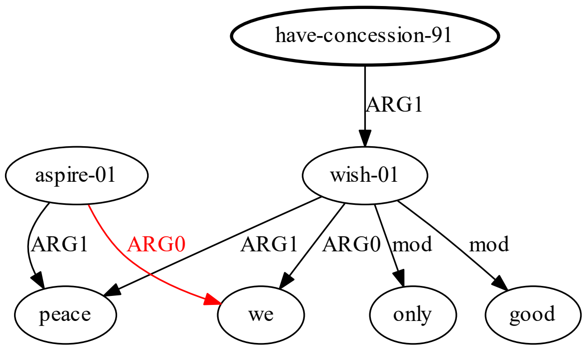
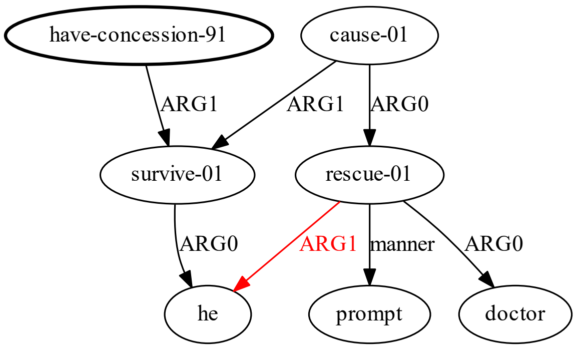


Figure 5.34: AMR for *The raven wants to learn.*

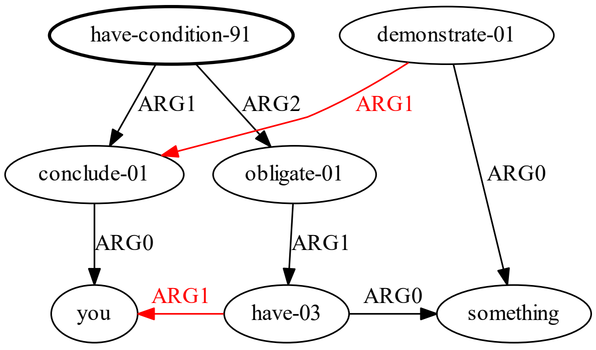
As we saw in the discussion in Section 5.4, there are some graphs where we cannot find a satisfactory analysis with the AM algebra. Largely, these are due to coreference. Given the constant set we defined in this section, we usually cannot obtain an analysis for these graphs at all, for example we do not obtain the constants used in Figure 5.24 in practice – which is a good thing, since that analysis was undesirable. There are many AMRs we cannot straight up parse with the AM algebra, about 34% of graphs in the LDC2015E86 training set (see the evaluation section below). We take a pragmatic approach here and simply remove all edges that cause a graph to have no AM term with the constants of



(a) *Nevertheless, the peace we aspire to will be only our good wish.*



(b) *However, with prompt rescue given by a doctor, he survived.*



(c) *To make a conclusion, you must have something to demonstrate it.*

Figure 5.33: Example graphs from the corpus where we remove an edges to cope with coreference and related problems (first five with seven nodes).

Section 5.5.1. This way, we can still use the rest of each graph for training.

We use the following algorithm for a given AMR. First, we check whether we can find an AM term with the algorithm defined in this section (i.e. whether the AM decomposition automaton we build has a non-empty language). If yes, we keep the graph unchanged. Otherwise, we use the linear representation of the AMR (as it is given in the original corpus) to find reentrant edges. For example, the graph in Figure 5.34 would have the representation

```
(w / want-0 :ARG0 (r / raven) :ARG1 (l/learn-01 :ARG0 r)).
```

Here, the second mention of the *raven* node `(r / raven)` in `(l/learn-01 :ARG0 r)` only uses the variable name `r`. This way of referring to a previously introduced edge with only its variable name is the indicator that the ARG0 edge from *learn-01* to *raven* is a reentrant edge.

At this point, we remove all reentrant edges from the graph, obtaining a tree. Then, we try to re-add all removed edges, one by one. Whenever we add an edge, we build a new decomposition automaton for the current graph. If its language is empty we discard the edge, otherwise we keep it.

This process gives us graphs that the AM algebra can analyze, in a reasonable amount of time. In the training set of the LDC2015E86 corpus, there is a total of about 290.000 edges in about 17.000 graphs. Of those edges, about 20.000 (ca. 7%) are reentrant. We remove about 12.000 edges with this procedure, i.e. ca. 60% of reentrant edges, or 4% of all edges.

Let us look at a few examples to get an idea of the effect of this in practice. Figure 5.33 shows three graphs from the corpus, where we removed edges with this procedure (the graphs are the first three of the graphs in the corpus with seven nodes and where we removed edges). The removed edges are printed in red. The graph in (a) indeed corresponds to a sentence with coreference:

(10) Nevertheless, the peace we$_i$ aspire to will be only our$_i$ good wish.

The removed edge, from *aspire-01* to *we*, corresponds to the first occurrence *we*$_i$ in the sentence.

The removed reentrancy in AMR (b) corresponding to

(11) However, with prompt rescue given by a doctor, he survived.

is not due to coreference, but due to secondary predication. We could find a proper term for this with the constant in Figure 5.35. However, our heuristics do not extract this constant, and it is unclear how to extend them to do this without obtaining too many undesirable terms elsewhere. In Figure 5.33(c), the ARG1 edge from *demonstrate-01* and *conclude-01* is again due to coreference, whereas the ARG1 edge from *have-03* to *you* is due to the conditional, again a phenomenon the AM algebra could model, assuming the right constants.

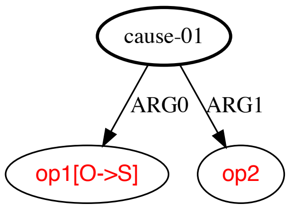


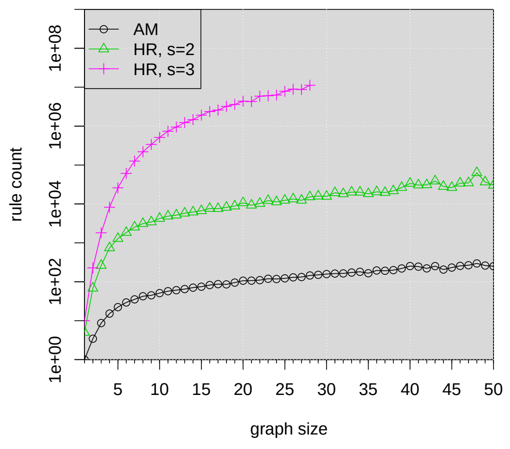
Figure 5.35: A constant to handle the graph in Figure 5.33(b).

This brief examination indicates that indeed, coreference is a significant cause of trouble to the AM algebra. But it also shows that a closer look at the phenomena in AMR and at our heuristics to extract the constants could be helpful in the future. However, many of these problematic cases are difficult and rare, lying on the Zipfian tail of language. Using a statistical approach to replace our heuristics for sources and annotations in the constants might be an approach to cover a larger range of phenomena without too much manual work. Finding a working method to handle coreference would also be desirable. But this is beyond the scope of this thesis. Here, this simple solution has good results: instead of losing 34% of the training instances, we only remove 4% of the edges.

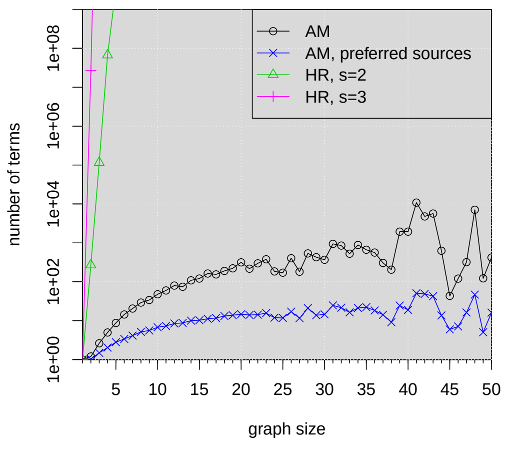
5.6 Evaluation

We conclude by analyzing whether the AM algebra achieves our goal of reducing the compositional complexity for a given AMR compared to the HR algebra, i.e. reducing the number of terms, automata sizes, and runtimes. Like in the previous chapter, we use all graphs of the LDC2015E86 training corpus with up to 50 nodes, for a total of 16616 graphs. We reuse the data from the final experiments of Chapter 4 for the comparison to the HR algebra. For the AM algebra, we use the method described in Section 5.5.

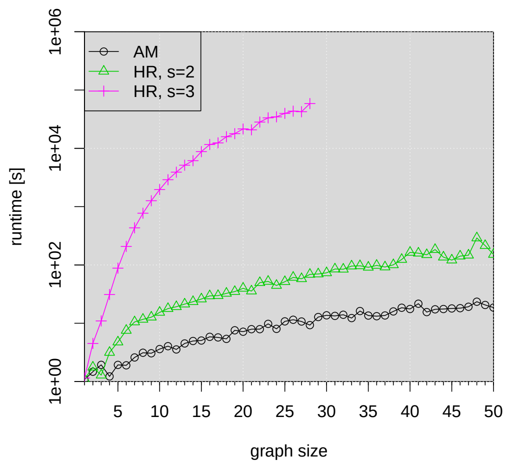
Automata size, and number of terms. The plots (a) and (b) of Figure 5.36 compare the automata sizes and language sizes for the AM algebra with the HR algebra with two and three sources. We find that the AM algebra achieves a dramatic reduction in both measures. For example, for the graphs with 15 nodes, the HR decomposition automata with three sources has automata with over a million rules, whereas the AM decomposition automata only have around 100 rules. Even for



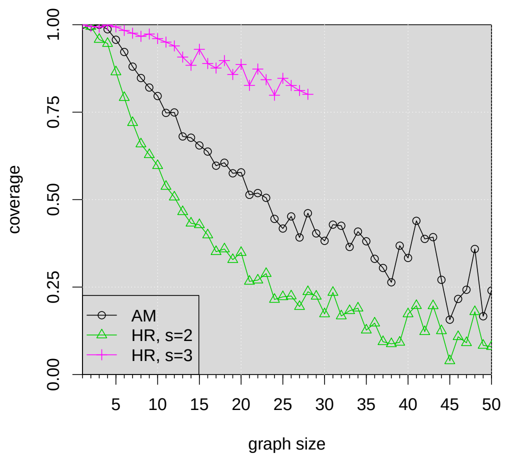
(a) Automata size



(b) Language size



(c) Runtimes



(d) Coverage

Figure 5.36: Evaluation on the LDC2015E86 dataset. For the HR algebra, this again uses the bottom-up algorithm, restricted to connected subgraphs and using atomic constants.

graphs with 50 nodes, the AM automata sizes are in the low hundreds. The difference in the number of terms in Figure 5.36(b) is even more drastic. Both HR algebra versions reach more than a billion terms for graph sizes in the single digits. The AM algebra, by contrast, has single digit language sizes there, and the averages remain below 10,000 across the dataset. The blue line shows the number of terms that achieve a maximal score according to our source preference model of Section 5.5.2. This gives another noticeable decrease in the number of terms, with now only about 10 terms remaining on average throughout the corpus.
These reductions have multiple reasons: we can use larger constants in the AM algebra, and the graph-combining operations of the AM algebra are much more constrained. Further, the type system and carefully chosen set of constants restrict application and modification.

Runtime. We also find significant improvements in terms of runtime. While the HR algebra with three sources couldn't finish the run through the dataset in several days, the AM algebra decomposed all graphs in a total of about two minutes. This makes a big difference when working with the method in practice. With two sources, the HR algebra can keep up a bit better than with three, but we saw before that a two source HR algebra is undesirable due to coverage issues, since it can only decompose trees.

Coverage. Consider now the coverage of the different graph algebras, i.e. the proportion of graphs of a given size for which we find at least one term, shown in Fig. 5.5.2(d). This reveals the problem discussed in Section 5.5.3: while the AM algebra's coverage is larger than that of the HR algebra with two sources – i.e. it can decompose some graphs that are not trees–, overall coverage is low at 66%. However, removing some (4%) of the reentrant edges as described in Section 5.5.3 changes the picture dramatically, see Figure 5.37. Coverage for the HR algebras increases slightly, but more importantly, the AM algebra now has nearly full coverage. The graphs that still cannot be decomposed mostly have a node in them, where our source assignment is forced to assign the same source to different nodes, which is illegal in the AM algebra. For example,

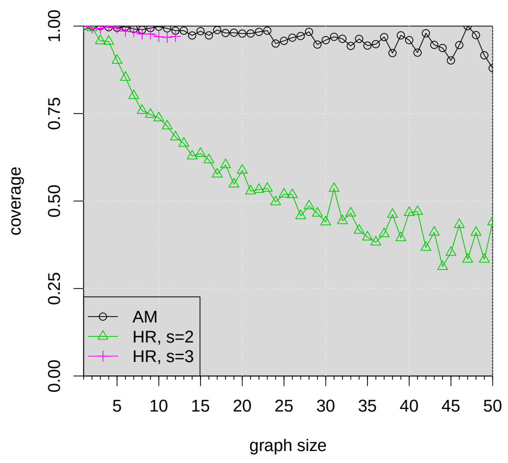


Figure 5.37: Coverage with edges removed according to Section 5.5.3.

when a node has two mod edges in its blob, both their blob targets get assigned the M source. A solution to this could be interesting future work, but the phenomenon is rare enough that we do not address it here. Overall, the AM algebra achieves a satisfying 98% coverage here.

5.7 Conclusion.

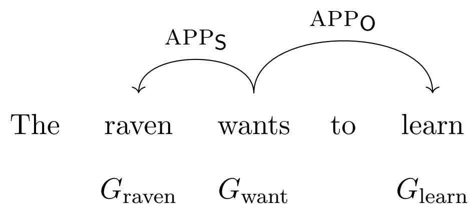
In this section we introduced the AM algebra, a typed graph algebra based on linguistic principles. It follows in the tradition of 'filling slots with arguments' we discussed in Chapter 3, using the mechanisms of application, modification and unification. We saw how the AM algebra can describe graphs resulting from complex linguistic phenomena, and at the same time simplifies the compositional analysis of a given graph. In the next section, we will use the AM algebra to build a compositional neural model for AMR parsing, yielding excellent results.

Future work. The need to remove edges as described in Section 5.5.3 is an important open problem. The examples given in that section illustrate that there is a variety of causes for removed edges, and a first step to resolving the problem would be a quantitative analysis of the causes. Possible steps in such an analysis include first to divide the causes into categories by looking at several examples, and then sorting a large amount of examples into these categories. The latter could be done manually or possibly by automatically matching patterns in the graphs. The differences between the categories may however depend on the string or be too subtle for automatic pattern matching to pick up, possibly making significant amounts of manual annotation necessary.

6

AM dependency parsing

In the previous chapter, we introduced the AM algebra and saw how it decreases the compositional complexity compared to the HR algebra. For a given AMR, we obtain much fewer AM terms than HR terms. At the same time, the AM algebra provides large coverage and linguistically reasonable terms. Now, we use the AM algebra to build an AMR parser.
We could at this point try to induce a grammar, such as an IRTG as seen in Chapter 2, using e.g. the sampling methods of Peng et al. (2015). However, this would probably lead to similar robustness and performance issues as for previous synchronous grammar models. Instead, we present a simpler and more robust *AM dependency* model. An AM dependency tree looks like this:



Some words in the sentence are associated with as-graph constants (written below the sentence), and a dependency tree describes the operations between them. AM dependency trees underspecify AM terms, and thus there are even fewer dependency trees describing an AMR than there are AM terms, further helping with obtaining consistent training data. At the same time, AM dependency trees do not underspecify the AMR. That is, when we obtain an AM term from the dependency tree and evaluate it to an AMR, that AMR is uniquely defined.

Further, and crucially, we can frame the prediction of an AM dependency tree as a supertagging plus a dependency parsing task. For both, we can use standard neural methods. This allows us to obtain a parser that has the best of both worlds: the

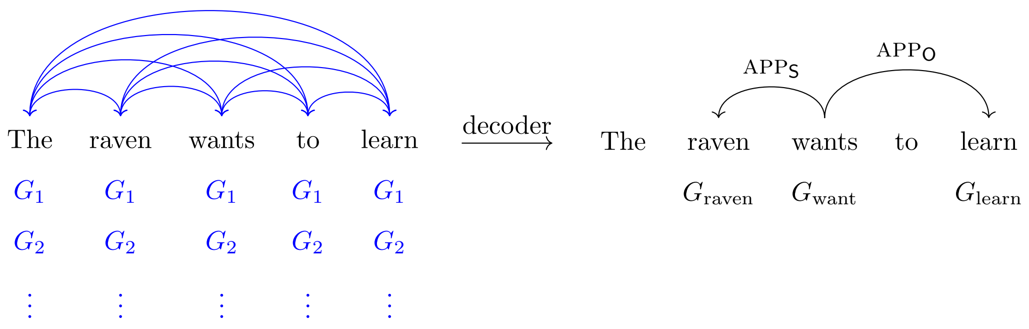


Figure 6.1: The neural model predicts scores for all possible edges and supertags, i.e. for all blue edges and supertags on the left. The decoder then finds the best AM dependency tree according to the scores.

linguistically motivated structure of the AM algebra to guide the model, combined with the robustness and effectiveness of neural networks.

We present the parsing model in Sections 6.1 to 6.7, with Section 6.1 giving an overview. Section 6.8 presents the evaluation results. In particular will we see that the AM dependency parser is competitive with the state of the art, and significantly outperforms a graph decoder baseline that has a near-identical neural network model, but lacks the structure of the AM algebra. To round off the chapter, Section 6.9 discusses limitations and strengths of the dependency approach, and ties back to the discussion in Chapter 3.

This chapter is based on the paper Groschwitz et al. (2018), and as in previous chapters, some text of that previous publication is reused here.

6.1 Model overview

We assume an edge and supertag factored model for the AM dependency parser (technical details in Section 6.3). That is, a neural network scores all possible edges (i.e. between each pair of words), and all supertags for each word. That is, scores for all blue edges and supertags in Figure 6.1 on the left. We then use a decoding step to find the best AM dependency tree according to these scores, under AM type constraints. The full pipeline has the following steps.

Training

1. Generate AM dependency trees for the AMRs in the training data. The AM dependency trees are described in Section 6.2, with implementation details on how we generate them in Section 6.7.

2. Train the neural supertagger and dependency models to predict the graph fragments and edges of the dependency trees in the training data (Section 6.4).

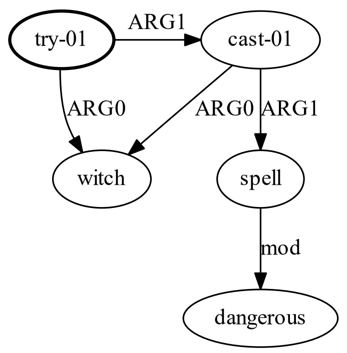
Prediction

1. Predict scored lists of potential graph fragments for each word, and scores for all possible edges with the neural models, see Figure 6.1 on the left.
2. Use a typed decoder (Section 6.5) to find the best well-typed dependency tree, according to the scores of the neural models, see Figure 1.6 on the right. We use a decoder with projectivity constraints, to make typed decoding tractable.
3. Evaluate the dependency tree to get an AMR, by first translating the dependency tree to an AM term, and evaluating that. Section 6.2 contains a proof that even though the AM term is underspecified, this evaluation process is well defined. Section 6.5 gives practical details.

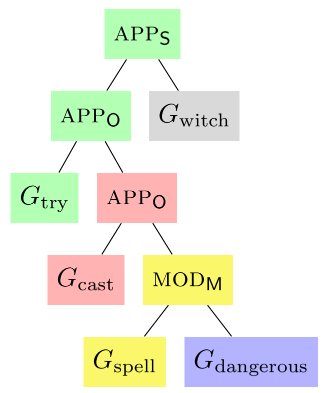
6.2 AM dependency trees

This section introduces AM dependency trees. We first discuss the intuition behind them as underspecifications of AM terms. Then, we provide a formal definition and prove that the dependency trees do in fact have a tree structure. A large part of this section is dedicated to the crucial proof that an AM dependency tree can be evaluated to a unique AMR, that is it does not underspecify the AMR, even though it underspecifies the AM term. We also discuss how underspecifying the AM term helps when obtaining structured training data.

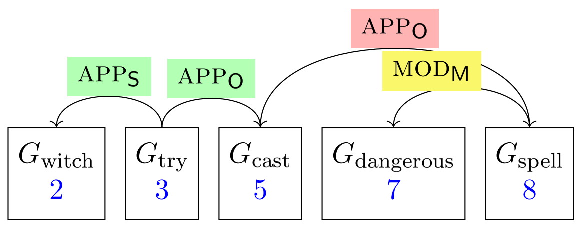
Figure 6.2(a) shows an AMR for *The witch tries to cast a dangerous spell.* In the corresponding AM term in (b), colors mark chains of operations that always follow the left child. Note that these chains start with a constant on the bottom left that creates a graph. Then the chains continue with operations that either fill argument slots of that graph or modify it. For example, the yellow MOD_{M} operation modifies G_{spell}, and the green APP_{O} and APP_{S} operations fill the argument slots of G_{try}. We call these chains of operations *maximal projections*, alluding to the related concept in X-bar theory. Essentially, the maximal projections track the *head* of the current subterm. On a technical level, this comes down to the fact that during such a maximal projection, the root source R remains at the same place, at the root of the head. That such a maximal projection follows the left child, i.e. is left-branching, fits with us calling the left child the head in both the apply and modify operations before.



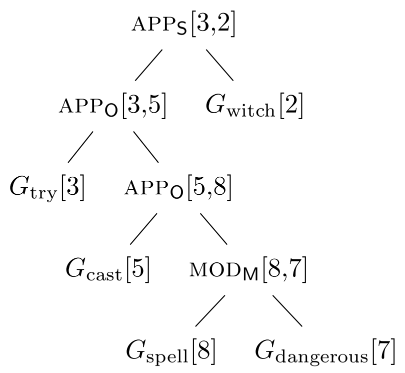
(a) AMR



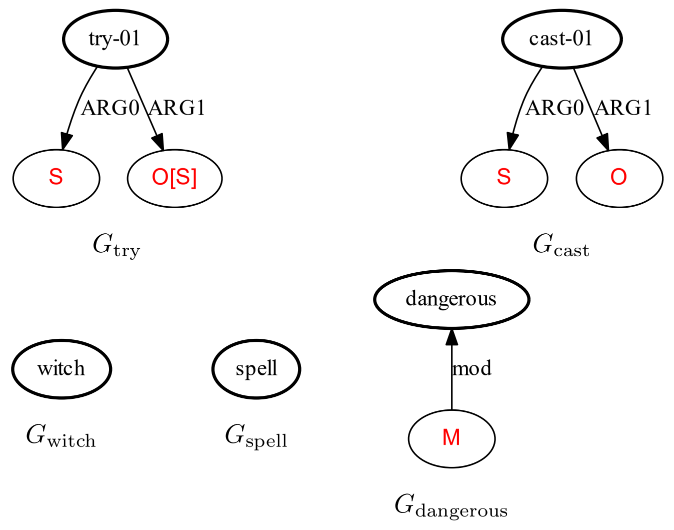
(b) AM term with maximal projections



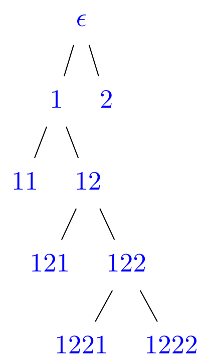
(c) AM dependency tree



(d) Indexed AM term



(e) Constants



(f) Positions in the term in (a)

Figure 6.2: (a) An AMR for *The witch tries to cast a dangerous spell*, (b) an AM term for it with the maximal projections marked in color, (c) a corresponding dependency tree and (d) an indexed AM term that produces (c). The used constants are shown in (e).

The AM dependency tree in (c) reflects that structure. Its nodes are labeled with the constants of the term, and the edges show the operations as relations between the heads. For example, the green $\textsc{app}_\mathsf{O}$ operation has head G_{try}, while the head of the argument is G_{cast}. Such a dependency tree underspecifies the order of operations in the AM term, for example here, the dependency tree does not specify which of the green $\textsc{app}_\mathsf{S}$ and $\textsc{app}_\mathsf{O}$ operations should come first. We will see below that this is not a problem, that while there may be many terms corresponding to a dependency tree, we can still assign a unique AMR to each dependency tree. But first, let us capture this intuition of heads and dependencies formally.

6.2.1 Definition

To define the AM dependency properly, we need to formalize the notion of the maximal projections, and we need to be able to follow each constant through the term for as long as it is the head. To keep track of each constant, we will associate the constants with indices that we track through the term, to obtain an *indexed AM term* as in Figure 6.2(d), with the indices in square brackets. These indices can be any natural number, but we will often use them to relate graph constants to word positions in the sentence. To formally define the indexing function, recall from Section 2.4.1 that we can view a term t as a function that assigns symbols to a set of *positions* $\mathsf{pos}(t)$. For example in Figure 6.2(b), the green $\textsc{app}_\mathsf{S}$ operation is at position ϵ, the green $\textsc{app}_\mathsf{O}$ at position 1, G_{try} at 11 and so on; the positions are shown in (f). We define an indexing function ind' on the leaves to be an injective function from the leaves of t to the natural numbers,

$$\mathsf{ind}' : \{p \in \mathsf{pos}(t) \mid p \text{ is a leaf in } t\} \to \mathbb{N}.$$

The injectivity here ensures that no two leaves have the same index assigned to them. In Figure 6.2(d), the indices are written at the leaves in square brackets.

We can then track the constants through the term by percolating their indices upwards. That is, we extend ind' to a function ind that is defined on all positions $p \in \mathsf{pos}(t)$. If p is a leaf, then simply $\mathsf{ind}(p) = \mathsf{ind}'(p)$. Otherwise, ind assigns to p a *pair* of indices (i, j); in Figure 6.2(d) again noted with square brackets next to the operations. In the pair (i, j), we call i the *head index* and j the *argument index*; at the leaves, we call the single number $\mathsf{ind}'(p)$ the head index. The idea is that if p is not a leaf, it must be a binary operation, and then we take the pair (i, j) where for i we use the head index assigned the left child, and for j the head index of the right child. This way, the index of a constant percolates along its maximal projection as the head index on the left. For example, the green maximal projection in (d) has the sequence $(G_{\text{try}}[3], \textsc{app}_{\mathsf{O}}[3,5], \textsc{app}_{\mathsf{S}}[3,2])$, keeping the index 3 on the left to indicate that G_{try} remains the head. Then, e.g. $\textsc{app}_{\mathsf{O}}[3,5]$ indicates that the argument head is the graph associated with index 5, namely G_{cast}.

Formally, we define a function head to be the identity on $\mathbb{N}$ and to map pairs (i,j) to i. That is, head maps a single index or a pair of indices to the head index. Then, given a term t and an indexing function ind' on its leaves, we define recursively, bottom-up, for all $p \in \mathsf{pos}(t)$,

$$\mathsf{ind}(p) = \begin{cases} \mathsf{ind}'(p) & \text{if } p \text{ is a leaf in } t \\ (\mathsf{head}(\mathsf{ind}(p1)), \mathsf{head}(\mathsf{ind}(p2))) & \text{otherwise.} \end{cases} \tag{6.1}$$

Because ind' is injective, ind is injective too: observe that every index assigned to a constant can be the right index in a pair, i.e. the argument index, only once. Then it stops being the head and is not percolated upward any further. Thus, ind cannot map two non-leaf positions to the same pair; injectivity on the leaves is inherited directly from ind'.

This concludes our definition of indexed AM terms:

Definition 6.1. Let t be an AM term and ind as in (6.1). Then we call the pair (t, ind) an *indexed AM term.*

We can now map indexed terms to dependency trees. In the dependency tree, we use the indices as nodes, label them with constants and obtain edges corresponding to the operations in the term. The percolated indices define the nodes for each edge. More precisely, using the square bracket notation of Figure 6.2(d), for every leaf $c[i]$ in the term we get a node i with label c in the dependency tree. For every operation $f[i,j]$ with head index i and argument index j, we get an edge from i to j with label f in the dependency tree. This way, the term in Figure 6.2(d) produces exactly the tree in (c) (the node identities in (c) are in blue). For example, the leaf $G_{\text{try}}[3]$ creates the node 3 with label G_{try} in the dependency tree, whereas the operation $\textsc{app}_{\mathsf{O}}[3,5]$ creates an edge from 3 to 5 with label $\textsc{app}_{\mathsf{O}}$. The following definition describes this mapping from indexed terms to dependency trees formally.

Definition 6.2 (Mapping to dependency trees)**.** Let (t, ind) be an indexed AM term. Then let $\mathsf{dep}(t, \mathsf{ind})$ be the simple graph T where for each position p, if p is a leaf, the index $\mathsf{ind}(p)$ is a node in T with label $t(p)$, i.e. the constant at p.
Otherwise, we have $\mathsf{ind}(p) = (i,j)$ for some indices i and j. Then there is an edge

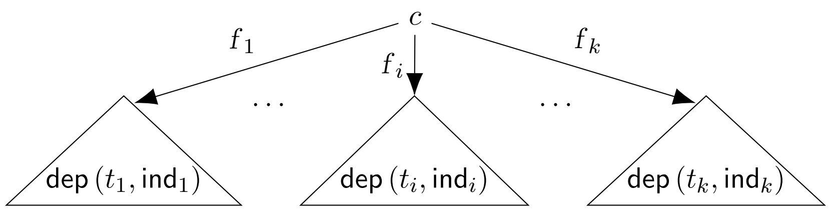

c
f_1
f_i
f_k
$\dots$
$\dots$
$\mathsf{dep}\,(t_1, \mathsf{ind}_1)$
$\mathsf{dep}\,(t_i, \mathsf{ind}_i)$
$\mathsf{dep}\,(t_k, \mathsf{ind}_k)$

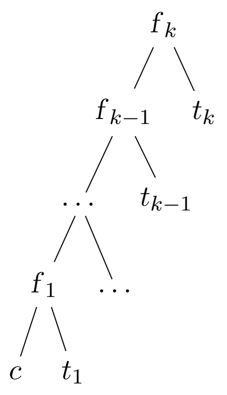
Figure 6.3: The recursive structure of a dependency tree $\mathsf{dep}\,(t, \mathsf{ind})$.

from i to j in T with label $t\,(p)$, i.e. from the head index to the argument index, with the operation that combines them as a label.
Formally and succinctly, we have $T = (V_T, E_T, \kappa_T, \lambda_T)$ (recall Definition 2.1 of simple graphs) with

$$\begin{aligned} V_T &= I\left(\mathsf{ind}'\right) \\ E_T &= I\,(\mathsf{ind}) \setminus I\left(\mathsf{ind}'\right) \\ \kappa_T &= t \circ \mathsf{ind}^{-1}\big|_{V_T} \\ \lambda_T &= t \circ \mathsf{ind}^{-1}\big|_{E_T} \end{aligned}$$

where we interpret t as a function from $\mathsf{pos}\,(t)$ to the signature.

We can take a recursive perspective on dependency trees that will be useful throughout this section. That is, we can characterize a dependency as a single node with edges going to nested dependency trees, as shown in Figure 6.3. To obtain this recursive structure, we first write a term t as



(6.2)

that is, we expand the topmost maximal projection. If we then apply the mapping dep to all the subterms $t_1, \dots, t_k$, we can make the following observation. The index

m_c assigned to the position of c by the indexing function ind is the head index throughout this left-branching term, this maximal projection. That is, at each f_i, the indexing function assigns a pair (m_c, j_i) for some index j_i. Visualized, we obtain the indices

$$\begin{tikzcd}
 & & & & f_k\,[m_c, j_k] \arrow[dl, dash] \arrow[dr, dash] & \\
 & & & f_{k-1}\,[m_c, j_{k-1}] \arrow[dl, dash] \arrow[dr, dash] & & t_k \\
 & & \dots \arrow[dl, dash] \arrow[dr, dash] & & t_{k-1} & \\
 & f_1\,[m_c, j_1] \arrow[dl, dash] \arrow[dr, dash] & & \dots & & \\
c\,[m_c] & & t_1 & & &
\end{tikzcd}$$

Furthermore, these j_i must be assigned to a leaf in t_i, and must thus be a node in the dependency tree for t_i. We can write the dependency tree for t_i as $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$ where ind_i is an indexing function corresponding to ind but defined on t_i only. We thus obtain the recursive structure for $\mathsf{dep}\,(t, \mathsf{ind})$ as shown in Figure 6.3. We obtain this structure because each index pair (m_c, j_i) corresponds to an edge from m_c to j_i with label f_i, and m_c has label c.

There is a technical issue with these indexing functions ind_i. To apply the function dep to the subterms t_i, we need to pair the t_i with indexing functions. But the indexing function ind is defined on the positions in t, not the positions in t_i. For example, the position ϵ of t_k is actually at the position 2 in t, the position ϵ of t_{k-1} is at 12 in t, ϵ of t_{k-2} is at 112 and so on. Thus, to get the appropriate indexing functions for the t_i we define 1_i to be the sequence of i ones. Then the position ϵ in t_i corresponds to the position $1_{k-i}2$ in t, and more generally, any position p in t_i corresponds to $1_{k-i}2p$ in t. We can then for each i define an indexing function ind_i that is consistent with ind but relative to t_i, by letting

$$\mathsf{ind}_i\,(p) = \mathsf{ind}\,(1_{k-i}2p)\,.$$

With these indexing functions ind_i we obtain the the recursive structure in Figure 6.3 properly, and can make the following formal statement to justify the figure.

Lemma 6.3. *Let t be as in 6.2, and ind an indexing function for t. Further let the ind_i be the just defined indexing structures on the t_i. Let $m_c = \mathsf{ind}\,(1_k)$ and*

$(j_i, \ell_i) = \mathsf{ind}_i\,(\epsilon)$. *Then*

$$\mathsf{ind}\,(1_{k-i}) = (m_c, j_i)$$

and $\mathsf{dep}\,(t, \mathsf{ind})$ *has the following structure.*

(i) *The node* m_c *has label* c.

(ii) *All* $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$ *are disjoint subgraphs of* $\mathsf{dep}\,(t, \mathsf{ind})$ *and there are no edges between them.*

(iii) *For every* $i = 1, \ldots, k$, *there is an edge from* m_c *to* j_i *with label* $\mathfrak{f}_i$, *and* j_i *is a node in* $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$. *There are no other edges incident to* m_c.

Proof. First, note that the statement $\mathsf{ind}\,(1_{k-i}) = (m_c, j_i)$ for all i follows directly from the facts that $m_c = \mathsf{ind}\,(1_k)$, $(j_i, \ell_i) = \mathsf{ind}_i\,(\epsilon) = \mathsf{ind}\,(1_{k-i}2)$ and the left head always percolates up. The statements (i-iii) follow immediately: Statement (i) is trivial. In (ii), that the $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$ are subgraphs of $\mathsf{dep}\,(t, \mathsf{ind})$ is a consequence of the definition of the ind_i, that they correspond to ind but relative to the subterms t_i. The disjointness in (ii) follows from the fact that ind is injective. That there are no edges between the subgraphs is because the subterms t_i are not directly connected in t, but only through the f_i. Finally, (iii) follows immediately from $\mathsf{ind}\,(1_{k-i}) = (m_c, j_i)$. $\square$

6.2.2 Tree structure

We referred to the graphs created by dep as *dependency trees*, and they indeed seem to be trees. But what exactly does that mean, being a tree? The following definition follows the standard definition of trees as connected, acyclic graphs. We also transfer the notion of an *origin* that we saw for DAGs in the previous chapter.

Definition 6.4 (Tree)**.** A (simple) graph $T = (V_T, E_T, \kappa_T, \lambda_T)$ is a *tree* if it is connected and has no undirected cycle, i.e. no non-trivial path $u_0 \leftrightarrow u_2 \leftrightarrow \ldots \leftrightarrow u_k$ with $u_0 = u_k$. Here "non-trivial" means $k \geq 1$; also recall that a path may not use the same edge twice.
A node $v \in V_T$ is an *origin*[1] of a tree T if for every node $u \neq v$, there is a directed path from v to u, $v \to u_1 \to \ldots \to u$.

A tree can be equivalently defined as a graph where between any two nodes, there is exactly one path (see e.g. Diestel (2018)). For an origin v then, the condition can

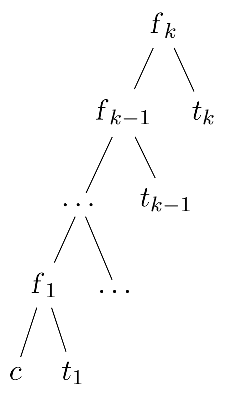
[1]As was the case for origins in DAGs: what I call origin here is often called a root or a source elsewhere.

be formulated as the unique path from v to each other node u being directed. This insight also implies that there can be only one origin. One can also easily see that a node v is the origin of a tree T if and only if v has no incoming edges, and any other node $u \neq v$ has exactly one incoming edge. We can use similar language for trees with origin as for terms, for example we can define a *leaf* as a node without outgoing edges, and the *depth* of a tree with origin as the longest directed path in it, i.e. the longest path from the origin to a leaf.

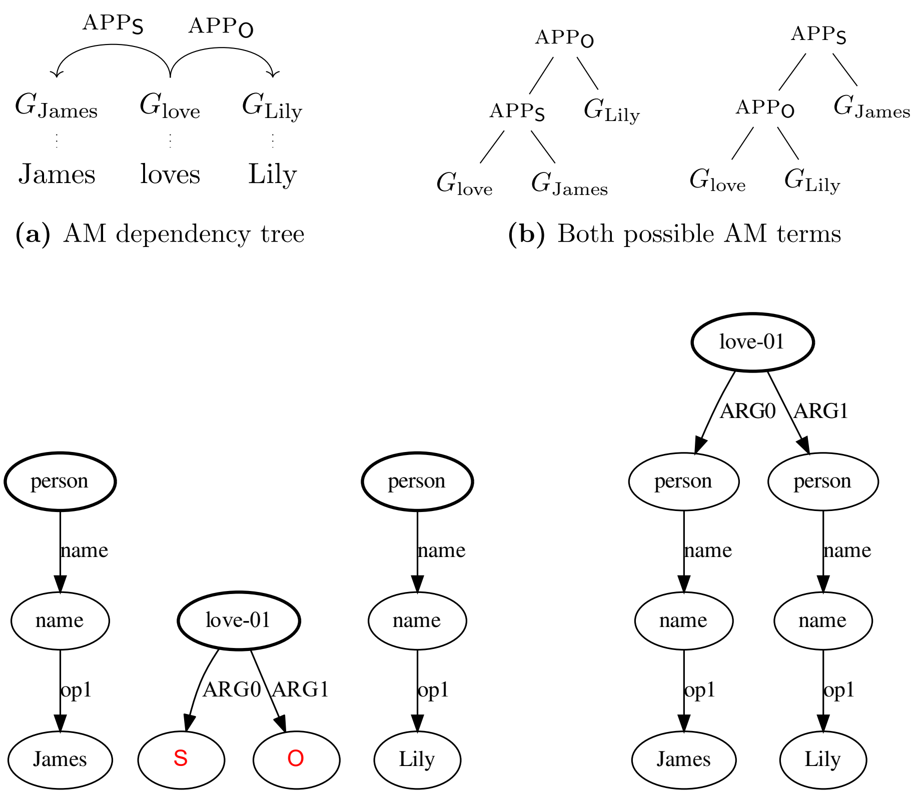
It seems indeed like the graphs created by dep are trees according to this definition, at least the graph in Figure 6.2 is. The recursive structure of Lemma 6.3 fits with this impression. Furthermore, in the example in Figure 6.2 the origin is the head index at the top of the term. Let us prove that these observations are always true.

Lemma 6.5. *For any indexed AM term* (t, ind)*, the dependency tree* $\mathsf{dep}(t, \mathsf{ind})$ *is indeed a tree, with origin the head index at* ϵ*, i.e.* $\mathsf{head}(\mathsf{ind}(\epsilon))$*.*

Proof. We show this via induction on the depth of t. If t is only a constant, then the statement is trivially true.
Otherwise, we again write t as



Each subterm t_i has lower depth than t, and with the indexing functions ind_i each pair (t_i, ind_i) is a proper indexed AM term. This means, we can conclude by induction that the dependency tree $\mathsf{dep}(t_i, \mathsf{ind}_i)$ is a tree with origin the head index of $\mathsf{ind}_i(\epsilon)$, i.e. the head index is j_i if $(j_i, \ell_i) = \mathsf{ind}_i(\epsilon)$. Further, let m_0 be the head index at the top of t. Then, in fact $m_0 = m_c$ with m_c the index at c as in Lemma 6.3. This is because the left index always gets percolated upwards as the head. Then, by Lemma 6.3, there is one edge from m_0 to each j_i, and no further edges at m_0.
We can now see that $\mathsf{dep}(t, \mathsf{ind})$ is a tree. Firstly, it is clearly connected. Further, all $\mathsf{dep}(t_i, \mathsf{ind}_i)$ are acyclic by induction and since all $\mathsf{dep}(t_i, \mathsf{ind}_i)$ have pairwise disjoint vertex sets and no direct edges between each other by Lemma 6.3, $\mathsf{dep}(t, \mathsf{ind})$ cannot



(a) AM dependency tree

(b) Both possible AM terms

(c) Constants G_James, G_love, G_Lily

(d) AMR

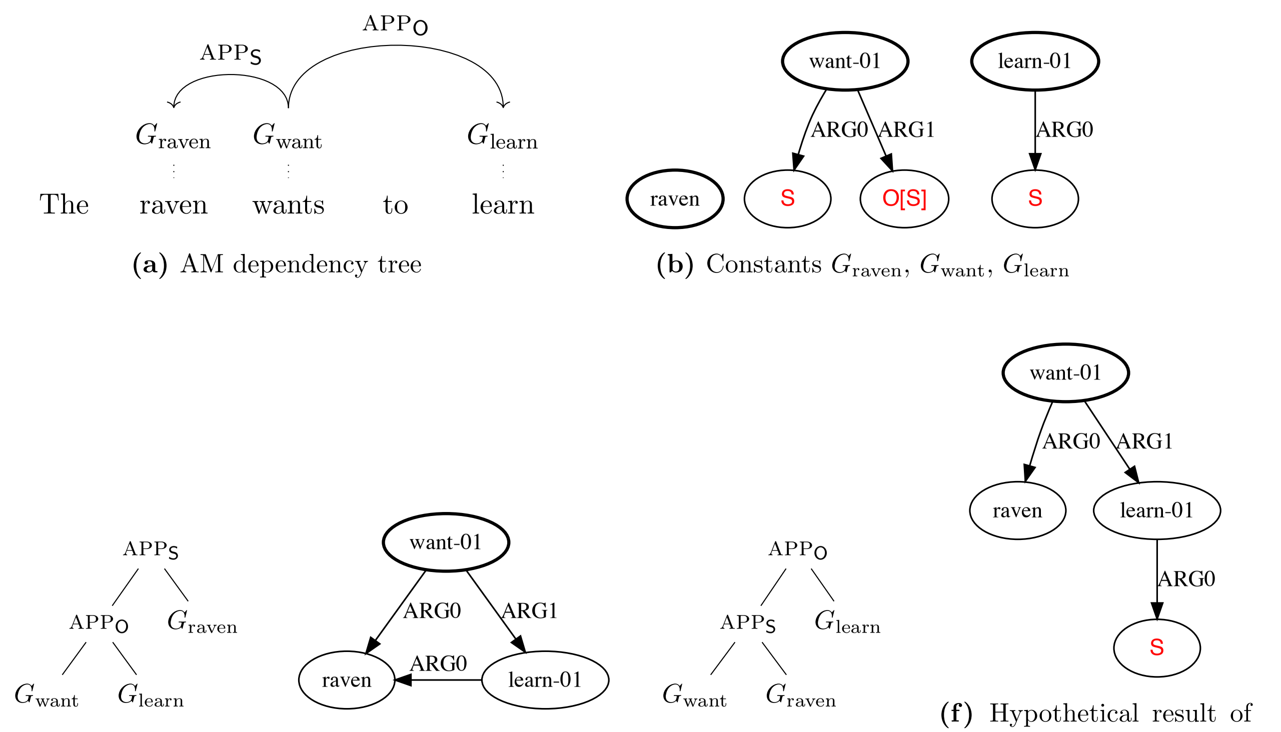
Figure 6.4: An AM dependency tree in (a), where both terms in (b) evaluate to the AMR in (d).

contain an undirected cycle either. Since we added exactly one incoming edge to the origin of each $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$, and m_0 has no incoming edge, m_0 is the origin of $\mathsf{dep}\,(t, \mathsf{ind})$. □

From Lemma 6.3 and Lemma 6.5 we in particular obtain that the j_i are the heads of the subtrees $\mathsf{dep}\,(t_i, \mathsf{ind}_i)$. Thus we have edges from the head of $\mathsf{dep}\,(t, \mathsf{ind})$ to the heads of the subtrees, labeled with the operations f_i. This gives a more complete understanding to the intuition shown in Figure 6.3.

6.2.3 Evaluating dependency trees

So far, we only described how to get an AM dependency tree from an AM term. But if our model predicts dependency trees, we also need to know how to evaluate them. So, let us consider a dependency tree T, i.e. T is a tree with origin, with AM constants as node labels and operations as edge labels. Firstly, we call a dependency tree T *well-typed* if there is a well-typed indexed term (t, ind) with $\mathsf{dep}\,(t, \mathsf{ind}) = T$.



(a) AM dependency tree

(b) Constants G_{raven}, G_{want}, G_{learn}

(c) Well-typed term

(d) AMR

(e) Ill-typed term

(f) Hypothetical result of the term in (e)

Figure 6.5: An AM dependency tree in (a), where only the term in (c) is well-typed. The ill-typed term (e) would evaluate to the wrong AMR (f).

We can then evaluate the term t to evaluate the dependency tree T. However, there may be many such well-typed terms t: the dependency trees define the operations, heads and arguments in a term, but underspecify the *order* of the operations.

For example, the dependency tree in Figure 6.4(a) states that there is an APP_{S} operation with head G_{love} and argument G_{James}, and an APP_{O} operation with head G_{love} and argument G_{Lily}, but it does not specify in which order they occur. Figure 6.4(b) shows both terms that satisfy these constraints, i.e. both terms that correspond to the dependency tree in (a). The dependency tree does not specify which of these terms to use, but here is the crucial observation that makes the dependency trees work: both terms in (b) evaluate to the same AMR in Figure 6.4(d). The graph G_{James} fills the S slot of G_{love} and G_{Lily} the O slot, no matter in which order.

The case is a bit different for the dependency tree in Figure 6.5(a). Here, the two possible terms in Figure 6.5(c) and (e) evaluate to different AMRs, shown in (d) and (f). However, only the term in (c) is well-typed (and it yields the correct AMR for the sentence). In the term in (e), the operation $\text{APP}_{\mathsf{S}}\,(G_{\text{want}}, G_{\text{raven}})$ violates Condition (i) of Definition 5.5, that S must be an origin in the type of the head (at this point in (e), S still has an incoming edge from O in $\tau\,(G_{\text{want}})$). In other words, this fills the S slot of G_{want} while it is still waiting for the unification from $\mathsf{O}[\mathsf{S}]$, which is not well-typed in the AM algebra. Since we only consider well-typed terms, the AM dependency tree describes a unique AMR in this case also.

What we do next is to show that these two examples correctly represent the general case, namely that given any well-typed dependency tree T, all corresponding well-typed terms t evaluate to the same AMR.

The idea of the proof is the following. We will see that if two terms t and s correspond to the same dependency tree, then they must be the same except for a possible reordering of operations inside maximal projections. We first show that such a reordering does not affect the evaluation result of the term as long as the term remains well-typed. We then conclude the proof below in Theorem 6.8.

First, we need to take a closer look at some properties of reordering maximal projections. Let (t, ind) be a well-typed indexed AM term such that

$$t \quad = \quad \begin{tikzcd}[column sep=small] & & & & f_k \arrow[dl, no head] \arrow[dr, no head] & \\ & & & f_{k-1} \arrow[dl, no head] \arrow[dr, no head] & & t_k \\ & & \dots \arrow[dl, no head] \arrow[dr, no head] & & t_{k-1} & \\ & f_1 \arrow[dl, no head] \arrow[dr, no head] & & \dots & & \\ c & & t_1 & & & \end{tikzcd}$$

We define a partial ordering $\prec_c$ on the pairs (f_i, t_i). The order depends on the constant c. We let

$$(f_i, t_i) \prec_c (f_j, t_j)$$

hold in the following cases.

(a) If $f_i = \text{APP}_\alpha$ and $f_j = \text{APP}_\beta$, and there is a directed path from α to β in the type of c, $\tau(\llbracket c \rrbracket)$.

(b) If $f_i = \text{MOD}_\alpha$ and $f_j = \text{APP}_\beta$, $\alpha \neq \beta$, and β is in the type of $\llbracket t_i \rrbracket$.

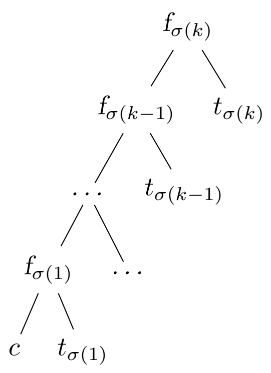
The idea is that reordering the operations and their right-hand side terms is OK as long as the partial order is respected. Essentially, Case (a) ensures that β is an origin in the type of the head when APP_β occurs. Case (b) ensures that when the modifier in MOD_α has an additional source β, then this β is still in the type of the head when MOD_α occurs. The other conditions on apply and modify are not affected by the reordering. The following lemma formalizes this.

Lemma 6.6. *Let t be a well-typed AM term such that*

$$t \quad = \quad \begin{tikzcd}[column sep=small] & & & & f_k \arrow[dl, no head] \arrow[dr, no head] & \\ & & & f_{k-1} \arrow[dl, no head] \arrow[dr, no head] & & t_k \\ & & \dots \arrow[dl, no head] \arrow[dr, no head] & & t_{k-1} & \\ & f_1 \arrow[dl, no head] \arrow[dr, no head] & & \dots & & \\ c & & t_1 & & & \end{tikzcd}$$

and let σ be a permutation on $\{1, \ldots, k\}$. Further, let

$s \quad =$



Then s is well-typed if and only if for all i, j with $(f_i, t_i) \prec_c (f_j, t_j)$, we have $\sigma(i) < \sigma(j)$.

Proof. $\underline{\Leftarrow}$: Let us first assume that the condition on the partial order holds, and show that s is well-typed. Recall the result from Section 5.3, that we can check well-typedness by looking only at the types, not the as-graphs themselves. Since all the subterms t_i are given as well-typed, we only need to check the operations $f_{\sigma(i)}$. Now consider how the type of the head changes as we go along the operations f_i. We start with the type of c itself, and at each $\textsc{app}_\alpha$ operation, the source α is removed. Modify operations leave the type unchanged.
In Section 5.3, we in particular saw that an operation $\textsc{app}_\alpha(\tau_1, \tau_2)$ succeeds if and only if

(i) τ_1, τ_2 are source dependency structures, i.e. $\tau_1, \tau_2 \neq \mathsf{FAIL}$.

(ii) α is an origin in τ_1, and

(iii) τ_2 matches the request of τ_1 at α, i.e. $\tau_2 = \mathsf{req}_{\tau_1}(\alpha)$.

Condition (i) is equivalent to the subterms below the operation being well-typed. We can guarantee this recursively, making our way up through s from $f_{\sigma(1)}$ to $f_{\sigma(k)}$. Condition (ii) means that α may not have incoming edges, which we guarantee with Case (a) of $\prec_c$. That is because, if there is a source β with a directed path to α in the type of c, respecting the partial order ensures that $\textsc{app}_\beta$ has consumed the β source before $\textsc{app}_\alpha$ occurs. Condition (iii) is not affected by σ.
Similarly, Section 5.3 states that a modify operation $\textsc{mod}_\alpha(\tau_1, \tau_2)$ is well-typed if and only if

(i) α is a root in τ_2,

(ii) $\mathsf{req}_{\tau_2}(\alpha) = (\emptyset, \emptyset, \emptyset)$, i.e. τ_2 does not have complex expectations at α, and

(iii) τ_2 without α is a subgraph of τ_1.

Conditions (i) and (ii) are not affected by σ. Since the only change to the type of the head (which here plays the role of τ_1) along the f_i operations is that sources are removed by application, we need to ensure that all sources in τ_2, minus α, are still there. If the $\textsc{mod}_\alpha$ operation is f_i, then $\tau_2 = \tau\left(\llbracket t_i \rrbracket\right)$ and this condition is guaranteed if Case (b) of $\prec_c$ being respected.
$\underline{\Rightarrow:}$ Conversely, let us assume that s is well-typed. If we assume that Case (a) of $\prec_c$ is violated in s, then the above Condition (ii) of $\textsc{app}_\alpha$ fails and s is not well-typed. If we assume that Case (b) of $\prec_c$ is violated, then Condition (iii) of the $\textsc{mod}_\alpha$ does not hold, and s is ill-typed again. □

The next lemma will allow us to reduce the case of general permutations to just the case of swapping two neighboring operations. It is a variation on the classic result that any permutation σ on a sequence of numbers $1, \ldots, k$ can be expressed through a sequence of *adjacent transpositions*, i.e. permutations on $1, \ldots, k$ that each only swaps a pair of neighboring numbers. That is, that there are adjacent transpositions $\mu_1, \ldots, \mu_m$ such that $\sigma = \mu_m \circ \ldots \circ \mu_1$. We show the proof in the appendix on page 234.

Lemma 6.7. *Let $\prec_c$ be a partial order on a finite set $x_1, \ldots, x_k$ and σ a permutation on $1, \ldots, k$, such that both the sequences $x_1, \ldots, x_k$ and $x_{\sigma(1)}, \ldots, x_{\sigma(k)}$ respect the partial order $\prec_c$; i.e. if $x_i \prec_c x_j$, then $i < j$ and $\sigma(i) < \sigma(j)$. Then there is a sequence of adjacent transpositions $\mu_1, \ldots, \mu_m$ such that $\sigma = \mu_m \circ \ldots \circ \mu_1$ and for every $\ell = 1, \ldots, M$, the intermediate sequence $x_{(\mu_\ell \circ \ldots \circ \mu_1)(1)}, \ldots, x_{(\mu_i \circ \ldots \circ \mu_1)(k)}$ respects $\prec_c$; i.e. if $x_i \prec_c x_j$ then $(\mu_\ell \circ \ldots \circ \mu_1)(i) < (\mu_\ell \circ \ldots \circ \mu_1)(j)$.*

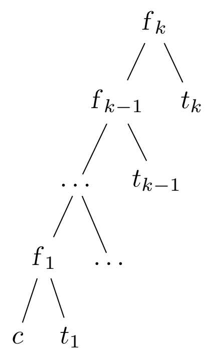
We can now show the central theorem of this section, that every well-typed dependency tree evaluates to a unique AMR.

Theorem 6.8. *Let T be a well-typed AM dependency tree and $\left(t, \mathsf{ind}^t\right)$ and $\left(s, \mathsf{ind}^s\right)$ be well-typed indexed AM terms such that*

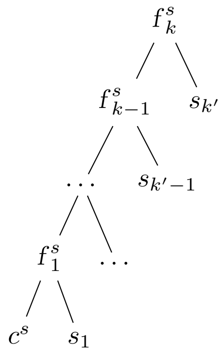
$$\mathsf{dep}\left(t, \mathsf{ind}^t\right) = \mathsf{dep}\left(s, \mathsf{ind}^s\right) = T.$$

Then t and s evaluate to the same as-graph, $\llbracket t \rrbracket = \llbracket s \rrbracket$.

Proof. We show the statement via induction on the depth of T.
We write t as



and s as



Since $\left(t, \mathsf{ind}^t\right)$ and (s, ind^s) define the same dependency tree T, by Lemma 6.5 they must have the same head index at their top (the origin of T), let us call it i_0. Since the label of i_0 in T must be equal to c, but also equal to c^s, in particular we must have $c = c^s$.
Further, by Lemma 6.3 we know that the operations f_i and f_i^s correspond to the outgoing edges of i_0 in T. More precisely, these edges are defined by the indices $\mathsf{ind}^t(1_i)$ and $\mathsf{ind}^s(1_j)$ in the two terms respectively. In particular, those sets of indices must overall be equal, and thus there is a permutation σ' such that for every $i = 1, \dots, k$ we have $\mathsf{ind}^s(1_i) = \mathsf{ind}^t\left(1_{\sigma'(i)}\right)$. Then, since $f_i = t(1_{k-i})$ and $f_i^s = s(1_{k-i})$, we get

$$f_i^s = s(1_{k-i}) = t\left(1_{\sigma'(k-i)}\right) = f_{k-\sigma'(k-i)}.$$

In the middle step, we used that the positions 1_{k-i} in t and $1_{\sigma(k-i)}$ in s correspond to the same edge in T, as we just saw, and must thus be identically labeled. To not get confused about the term $k - \sigma'(k-i)$ in the end, let us simply define a new permutation σ via $\sigma(i) = k - \sigma'(k-i)$. We thus have $f_i^s = f_{\sigma(i)}$ and can write

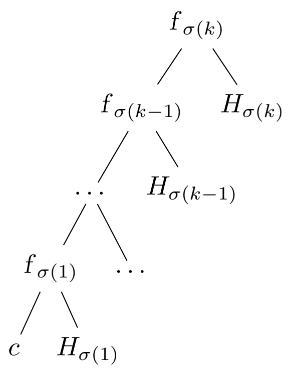
$$s = \begin{tikzcd}[column sep=small]
 & & & & f_{\sigma(k)} \arrow[dl, dash] \arrow[dr, dash] & \\
 & & & f_{\sigma(k-1)} \arrow[dl, dash] \arrow[dr, dash] & & s_k \\
 & & \dots \arrow[dl, dash] \arrow[dr, dash] & & s_{k-1} & \\
 & f_{\sigma(1)} \arrow[dl, dash] \arrow[dr, dash] & & \dots & & \\
c & & s_1 & & &
\end{tikzcd}$$

Further, using the method to create ind_i above to obtain indexing functions ind_i^t and ind_i^s, Corollary 6.3 implies that the dependency trees $\mathsf{dep}\left(s_i, \mathsf{ind}_i^s\right)$ and $\mathsf{dep}\left(t_{\sigma(i)}, \mathsf{ind}_{\sigma(i)}^t\right)$ are actually the same subtree of T. This is because we set up σ to be consistent with the indices in the terms, which determine the edge in T, which determine the subtree in T.
Now, in fact, these subtrees $\mathsf{dep}\left(s_i, \mathsf{ind}_i^s\right) = \mathsf{dep}\left(t_{\sigma(i)}, \mathsf{ind}_{\sigma(i)}^t\right)$ have a lower depth than T, and therefore we can use the induction hypothesis to conclude that the terms s_i and $t_{\sigma(i)}$ must evaluate to the same as-graph, $[\![s_i]\!] = [\![t_{\sigma(i)}]\!]$. Let us write $H_i = [\![t_i]\!] = [\![s_{\sigma^{-1}(i)}]\!]$ for this as-graph; in particular we have $[\![s_i]\!] = H\sigma(i)$.
All together, we now know that the following term[2] evaluates to the same graph as t:

$$\begin{tikzcd}[column sep=small]
 & & & & f_k \arrow[dl, dash] \arrow[dr, dash] & \\
 & & & f_{k-1} \arrow[dl, dash] \arrow[dr, dash] & & H_k \\
 & & \dots \arrow[dl, dash] \arrow[dr, dash] & & H_{k-1} & \\
 & f_1 \arrow[dl, dash] \arrow[dr, dash] & & \dots & & \\
c & & H_1 & & &
\end{tikzcd}$$

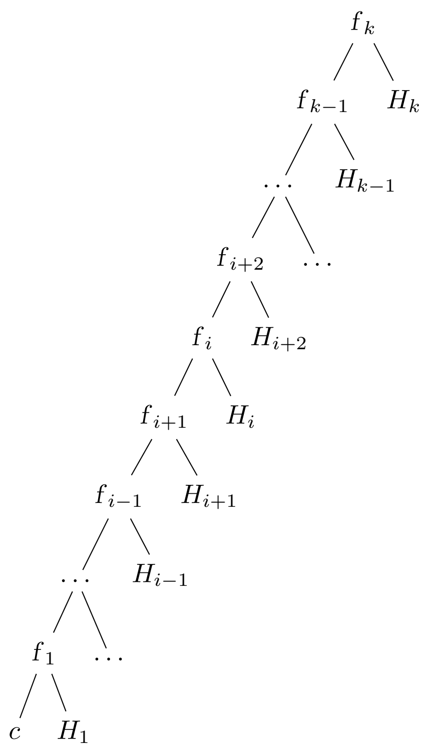
and the following term evaluates to the same graph as s:

[2]Technically, we need to change our algebra here, by adding $H_1, \dots, H_k$ as constants. However, when adding constants to an algebra $\mathcal{A}$ to obtain an algebra $\mathcal{A}'$, the new algebra $\mathcal{A}'$ still evaluates old terms over $\mathcal{A}$ to the same values. Thus, we can simply add constant symbols for $H_1, \dots, H_k$ to our algebra here, without invalidating the proof.



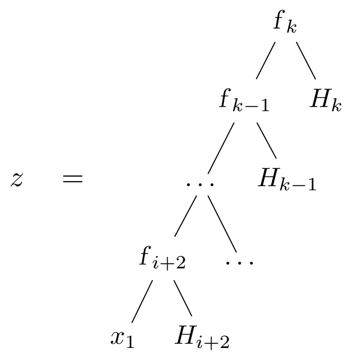
Using Lemmas 6.6 and 6.7 together, we can then see that there is a sequence of adjacent transpositions $\mu_1, \dots, \mu_m$ that together form σ, i.e. $\mu_m \circ \dots \circ \mu_1 = \sigma$, and at each step in between, the intermediary result, i.e. applying $\mu_\ell \circ \dots \circ \mu_1$ for any ℓ, gives a well-typed term. It is thus sufficient to show that any such adjacent transposition between two well-typed terms leaves the result unchanged. Then, if none of the adjacent transpositions change the evaluation result, neither will the full permutation.

That is, we can reduce the theorem to the case where s evaluates to the same graph as the term

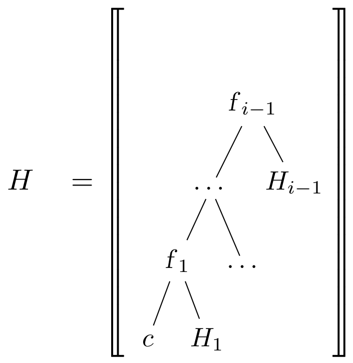


that is, just the order of i and $i+1$ is switched.

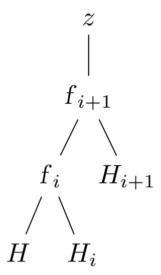
If we now write z for the context



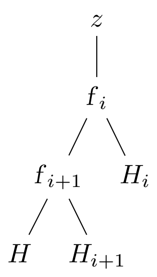
and write H for the evaluation result



then we only need to consider the case where t is



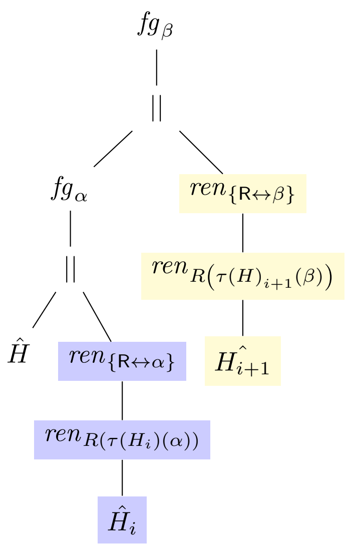
and s is



Since the context z is the same in both cases, we only have to show that

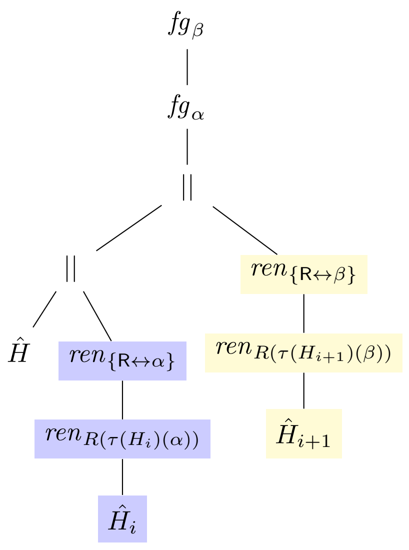
$$[\![f_{i+1}\left(f_i\left(H, H_i\right), H_{i+1}\right)]\!] = [\![f_{i+1}\left(f_i\left(H, H_{i+1}\right), H_i\right)]\!]$$

We need to check this for the three cases where both of the operations $f_i, f_i + 1$ are application, both are modification, or there is one of each. It is easy to see that changing the order of operations (as long as both are well-typed) does not change the type of the outcome. What remains is to check what happens to the s-graphs involved. Simply using the HR term from the definition of the apply operation, we obtain t in the case where $f_i = \text{APP}_\alpha$ and $f_{i+1} = \text{APP}_\beta$ the term


fg_β
$||$
fg_α
$ren_{\{\mathsf{R}\leftrightarrow\beta\}}$
$||$
$ren_{R\left(\tau(H)_{i+1}(\beta)\right)}$
$\hat{H}$
$ren_{\{\mathsf{R}\leftrightarrow\alpha\}}$
$\hat{H_{i+1}}$
$ren_{R(\tau(H_i)(\alpha))}$
$\hat{H}_i$

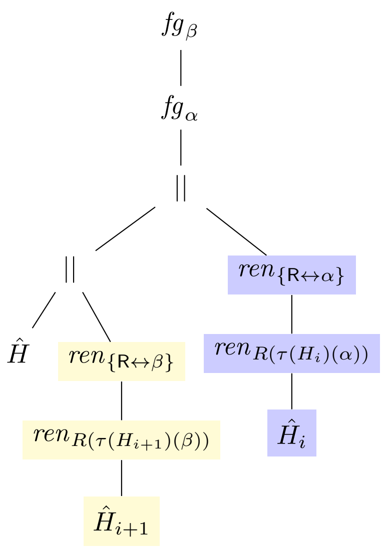
with some colors added so we can refer to the subterms conveniently. From how the source annotations are designed, the sources in the s-graph produced by the blue subterm are all among the sources to which there is a directed path from α in $\tau\left(H\right)$. Similarly, the sources in the s-graph produced by the yellow subterm are all among the sources reachable from β in $\tau\left(H\right)$. Since both orderings of APP_α and APP_β are allowed, we know that there is no directed path from α to β in $\tau\left(H\right)$ nor the other way around. In conclusion, the s-graph produced by the blue term does not contain a β source, and the s-graph produced by the yellow term does not contain an α source.

We can therefore delay the fg_α operation until after the topmost merge – since the right-hand child of the merge does not contain a α source, this does not change the outcome. We obtain the term



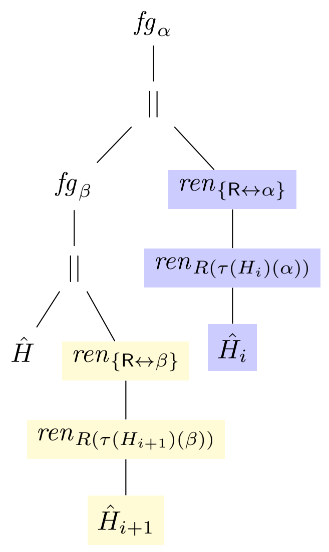
fg_β
fg_α
$||$
$||$
$ren_{\{\mathsf{R}\leftrightarrow\beta\}}$
$\hat{H}$
$ren_{\{\mathsf{R}\leftrightarrow\alpha\}}$
$ren_{R(\tau(H_{i+1})(\beta))}$
$ren_{R(\tau(H_i)(\alpha))}$
$\hat{H}_{i+1}$
$\hat{H}_i$

We can now swap the order of the merge operations (Courcelle and Engelfriet (2012) rightly note that the merge operation is associative) to obtain



fg_β
fg_α
$||$
$||$
$ren_{\{\mathsf{R}\leftrightarrow\alpha\}}$
$\hat{H}$
$ren_{\{\mathsf{R}\leftrightarrow\beta\}}$
$ren_{R(\tau(H_i)(\alpha))}$
$ren_{R(\tau(H_{i+1})(\beta))}$
$\hat{H}_i$
$\hat{H}_{i+1}$

Finally, with the same argument as delaying the fg_α operation earlier we can move the fg_β operation downwards, obtaining the term


fg_α
$||$
fg_β
$ren_{\{\mathsf{R}\leftrightarrow\alpha\}}$
$||$
$ren_{R(\tau(H_i)(\alpha))}$
$\hat{H}$
$ren_{\{\mathsf{R}\leftrightarrow\beta\}}$
$\hat{H}_i$
$ren_{R(\tau(H_{i+1})(\beta))}$
$\hat{H}_{i+1}$

which indeed corresponds to s in this case. The cases for two modify operations, and one apply and one forget, can be treated similarly. □

With this proof of Theorem 6.8 complete, we now know that every well-typed AM dependency tree evaluates to a unique, well-defined AMR. Thus, predicting an AM dependency tree is a valid approach to an AMR parser. We will discuss the model in Section 6.3. But first, we have a look at how the dependency trees contribute to solving the training data issue we worked on in the previous two chapters.

6.2.4 Reduced ambiguity in obtaining AM dependency trees

We saw in the last chapter how the AM algebra reduces the set of possible terms for a given AMR G drastically compared to the HR algebra. Now, multiple AM terms map to the same AM dependency tree, which means that there are even fewer AM dependency trees that describe G. In fact, observations made when working on the dataset lead me to make the following conjecture: that if we fix which as-graph constants we use for which parts of G, and fix the indexing function ind, then there is at most one AM dependency tree remaining. That is, the only ambiguity left is what constants we choose.

To formalize this, we can again use concrete as-graphs – this allows us to properly express the part of "which as-graph constants are used for which parts of G". Recall the concrete AM algebra we used in Section 5.2.2, with domain the concrete as-graphs. Each constant in a concrete AM term describing a concrete representative

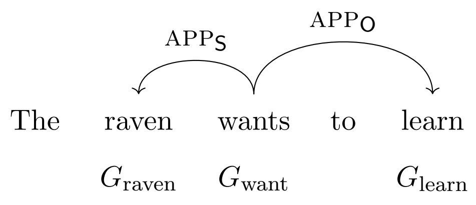


Figure 6.6: AM dependency tree for *The raven wants to learn.*

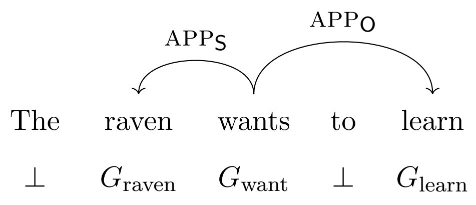


Figure 6.7: AM dependency tree for *The raven wants to learn*, including empty supertags $\perp$.

of G is a subgraph of that representative; in fact the constants in such a term partition the representative's edge set. We can straightforwardly extend the notion of an indexed AM term, and thus an AM dependency tree, to the concrete AM algebra. Each such concrete term or dependency tree can in turn be translated back to the standard (i.e. abstract) version by replacing each concrete constant with its equivalence class under isomorphism. I conjecture the following statement.

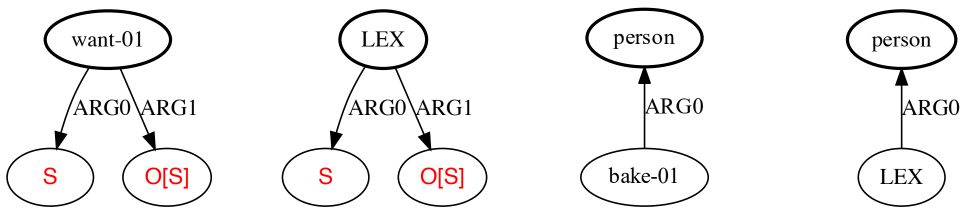
Conjecture 6.9. *Let* (t_1, ind_1) *and* (t_2, ind_2) *be concrete indexed AM terms, such that* $[\![t_1]\!] = [\![t_2]\!]$*,* t_1 *and* t_2 *use the same (concrete) constants, and* ind_1 *and* ind_2 *map leaves with the same constants to the same indices. Then the dependency trees* $\mathsf{dep}\,(t_1, \mathsf{ind}_1)$ *and* $\mathsf{dep}\,(t_2, \mathsf{ind}_2)$ *are the same.*

This concludes our formal examination of AM dependency trees. The next chapter introduces the AM dependency parsing model.

6.3 The parsing model

In the previous section, we established that an AM dependency tree uniquely describes an AMR. Now, we describe an AMR parsing model that translates a sentence to an AM dependency tree, and evaluates that to an AMR.

First, we connect an AM dependency tree $T = (V_T, E_T, \kappa_T, \lambda_T)$ to a sentence $\mathbf{w} = w_1 w_2 \ldots w_n$, by interpreting the indices that are the nodes in V_T as word



(a) G_{want} **(b)** G_{want} delexicalized **(c)** G_{baker} **(d)** G_{baker} delexicalized

Figure 6.8: Two graph constants (a,c) and their delexicalized versions (b,d)

positions in the sentence $\mathbf{w}$, an example is shown in Figure 6.6. Thus, we can interpret the node labeling function κ_T as a function that assigns a constant symbol, i.e. a graph fragment, to some words in $\mathbf{w}$. We call these graph fragments *supertags*, in reference to the supertagging technique for grammars that predicts lexicalized rules for each word. The supertags are written below the sentence in Figure 6.6. The edges of T form a dependency tree over parts of the sentence.

Our model predicts supertags for each word. Since not every word has a constant symbol associated with it (e.g. *The* and *to* in Figure 6.6), we allow the supertagger to also predict the empty tag $\bot$, see Figure 6.7.

We can now formulate the task of AMR parsing as the task of predicting the highest scoring AM dependency tree T with nodes in $\{1, \ldots, n\}$, according to some scoring function ω. Here we assume a node-factored (i.e. supertag-factored) and edge-factored model for the score $\omega(T)$ of the AM dependency tree T:

$$\omega(T) = \sum_{i \in V_T} \omega_{[i]}(\kappa_T(i)) + \sum_{i \in \{1,\ldots,n\} \setminus V_T} \omega_{[i]}(\bot) + \sum_{(i,j) \in E_T} \omega_{[i \to j]}(\lambda_T((i,j))), \quad (6.3)$$

where $\omega_{[i]}$ are scoring functions at each word position, and $\omega_{[i\to j]}$ are scoring functions for each possible edge. The sum $\sum_{i\in\{1,\ldots,n\}\setminus V_T} \omega_{[i]}(\bot)$ computes the scores of empty supertags for all word positions not used in the dependency tree. We decompose the edge weight further into the sum $\omega_{[i\to j]}(f) = \omega^{\mathsf{ex}}_{[i\to j]} + \omega^{\mathsf{lbl}}_{[i\to j]}(f)$ of a score $\omega^{\mathsf{ex}}_{[i\to j]}$ for the presence of an edge from i to j and a score $\omega^{\mathsf{lbl}}_{[i\to j]}(f)$ for this edge having label f.

We further assume that every graph constant corresponding to a word has a unique 'lexicalized' node whose label corresponds most closely to the word. For example, in Figure 6.8(a), we have the graph G_{want} corresponding to word *want*, where the *want-01* node matches the word. In G_{baker} in Figure 6.8(c), the graph corresponding to the word *baker*, that node is the *bake-01* node. We will see in Section 6.7 that this holds up in practice. We can thus interpret a graph constant as a pair of a 'delexicalized' graph fragment where the lexicalized node label is replaced with a `LEX` marker (see Figure 6.8(b,d)), and the replaced label. E.g. instead of G_{want} in Figure 6.8(a), we predict the delexicalized graph in Figure 6.8(b) and the label *want-01*. We factor the node weight such that the delexicalized graph and the lexical label are predicted separately by weights $\omega^{\mathsf{g}}_{[i]}$ and $\omega^{\mathsf{lbl}}_{[i]}$ respectively, i.e. $\omega_{[i]}(c) = \omega^{\mathsf{g}}_{[i]}(c) + \omega^{\mathsf{lbl}}_{[i]}(c)$. Splitting the constants in labels and delexicalized graph fragments greatly reduces the vocabulary size for the supertagger (from 28730 lexicalized graph fragments to 2370 delexicalized graph fragments) and allows it to generalize the use of graph fragments over lexicon entries. This is particularly helpful when dealing with rare or unseen words at evaluation time, see Section 6.7 below.

To recap, our aim is now to compute the well-typed tree T with the highest score $\omega(T)$. The next section describes how we obtain the scoring function ω in practice.

6.4 Training

We use recurrent neural networks to compute the scoring function ω. I present a short technical introduction to neural network first, and then our neural AM dependency model.

6.4.1 Neural network background

The idea of a neural network is to combine layers of linear and non-linear functions on real-valued vectors. A typical linear function is an affine transformation

$$A(\vec{x}) = W\vec{x} + \vec{b}$$

where the matrix W and the bias vector $\vec{b}$ are parameters of A. Typical non-linear functions are applied to a vector element-wise, and include

- tanh, which takes values between -1 (for input approaching negative infinity) and 1 (for input approaching infinity),
- the *sigmoid* function $\sigma(x) = \frac{1}{1+e^{-x}}$, which is shaped similarly to tanh but takes values between 0 and 1, and
- the rectified linear unit ReLU with $\mathrm{ReLU}(x) = \max(0, x)$.

A classic neural network layer would then have e.g the shape $\tanh(A(\vec{x}))$ for some affine transformation A. Such layers can be stacked on top of each other to obtain more powerful networks. For computing the output of a layer, one often uses the *softmax* function with

$$\mathrm{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}.$$

These entries of the softmax function sum to 1 and are non-negative. Thus, using a softmax layer with a K-dimensional input $\vec{x}$ allows computing a probability distribution for K discrete choices. A neural network is then trained using the backpropagation algorithm, using gradient descent to minimize a chosen *loss function*. The loss function is defined at the output layer, and the gradient is propagated back through the network.

LSTM. A *Long Short-Term Memory* recurrent neural network, or LSTM, is a recurrent network to encode sequences of vectors. These sequences here are sentences, with the words represented as vectors (details on that below). At each step t through the sequence, the LSTM makes several computations. It uses the input $\vec{x}_t$ to compute a *hidden state* $\vec{h}_t$ and a *cell state* $\vec{c}_t$, also using the hidden state $\vec{h}_{t-1}$ and (to a lesser extend) the cell state $\vec{c}_{t-1}$ of the previous step. The cell state $\vec{c}_t$ serves as a more long term memory, whereas the hidden state $\vec{h}_t$ is more localized, and is also used as the output at step t. We always initiate $\vec{h}_0$ and $\vec{c}_0$ as all zeros.
To compute the new states $\vec{h}_t$ and $\vec{c}_t$, the LSTM first combines the input and the last hidden state into a new vector:

$$\vec{g}_t = \tanh\left(A_{xg}\left(\vec{x}_t\right) + A_{hg}\left(\vec{h}_{t-1}\right)\right)$$

where A_{ig} and A_{hg} are affine transformations. The LSTM then computes several *gates*, which are vectors that will control the flow of information through the LSTM.

$$\begin{aligned}
\vec{i}_t &= \sigma\left(A_{xi}\left(\vec{x}_t\right) + A_{hi}\left(\vec{h}_{t-1}\right)\right)\\
\vec{f}_t &= \sigma\left(A_{xf}\left(\vec{x}_t\right) + A_{hf}\left(\vec{h}_{t-1}\right)\right)\\
\vec{o}_t &= \sigma\left(A_{xo}\left(\vec{x}_t\right) + A_{ho}\left(\vec{h}_{t-1}\right)\right)
\end{aligned}$$

Here $\vec{i}_t$ is called the *input gate*, $\vec{f}_t$ the *forget gate* and $\vec{o}_t$ the *output gate*, note that these all take values between 0 and 1 due to the sigmoid function. This allows computing the new cell and hidden states:

$$\vec{c}_t = \vec{f}_t\vec{c}_{t-1} + \vec{i}_t\vec{g}_t$$

(using the element-wise product), that is the forget gate $\vec{f}_t$ determines which entries of the old cell state $\vec{c}_{t-1}$ are carried over, and the input gate $\vec{i}_t$ determines which entries of the new information $\vec{g}_t$ are committed to memory. The new hidden state is then

$$\vec{h}_t = \vec{o}_t\vec{c}_t,$$

that is the output gate $\vec{o}_t$ controls which entries of $\vec{c}_t$ are pulled out of memory to be used as the local output, and as the context for the next LSTM step.

For a sequence of vectors $\vec{\mathbf{x}} = (\vec{x}_1, \dots, \vec{x}_n)$, and a fixed LSTM, we write $\mathrm{LSTM}(\vec{\mathbf{x}})_t$ for the output of the LSTM at step t, i.e. the hidden state h_t.

BiLSTM. For a sequence of vectors $\vec{\mathbf{x}} = (\vec{x}_1, \dots, \vec{x}_n)$, let the *forward sequence* $\vec{\mathbf{x}}^+$ be the original sequence $\vec{\mathbf{x}}$, and let the *backward sequence* $\vec{\mathbf{x}}^-$ be the sequence in reverse order. By combining an LSTM on the forward sequence with an LSTM on the backward sequence, we obtain a *bidirectional LSTM*, or BiLSTM, that takes context from both sides into account. Let us denote the two LSTMs with LSTM^+ and LSTM^-, then we have

$$\mathrm{BiLSTM}(\vec{\mathbf{x}}, t) = \mathrm{LSTM}^+\left(\vec{\mathbf{x}}^+, t\right) \circ \mathrm{LSTM}^-\left(\vec{\mathbf{x}}^-, n+1-t\right)$$

Note that by reversing the sequence and having 1-based indices for the step t, the result of $\mathrm{LSTM}^-(\vec{\mathbf{x}}^-, n+1-t)$ in fact corresponds to the t-th entry of the original sequence.
We can also stack these BiLSTMs, computing a first BiLSTM_1 as normal on the input sequence $\vec{\mathbf{x}}$, and then using the sequence $(\mathrm{BiLSTM}_1(\vec{\mathbf{x}}, t))_{t=1,\dots,n}$ as input for a second BiLSTM_2. We can interpret this as follows. The first BiLSTM_1 gets as input the general, out-of-context meaning of the words. It then contextualizes this meaning. When the stacked BiLSTM_2 then computes the desired output, it has access to contextualized word meanings. Additionally, using two stacked BiLSTMs simply allows for learning more complex interactions. Kiperwasser and Goldberg (2016) for example use this architecture for syntactic dependency parsing, and we follow their lead here.

6.4.2 The neural AM dependency model

We present two models for ω: one for the graph scores $\omega_{[i]}$ and one for the edge scores $\omega_{[i \to j]}$. All of these are based on the two-layer BiLSTM,[3] which reads a sequence of input vectors $\vec{\mathbf{x}} = (\vec{x}_1, \dots, \vec{x}_n)$ and produces vector representations $\vec{v}_i = \mathrm{BiLSTM}_2(\vec{\mathbf{x}}, i)$ for the individual input tokens (see Fig. 6.9). The two models differ in the inputs $\vec{\mathbf{x}}$ and the way they predict scores from the v_i. The PyTorch implementation of this model is open source, and can be found at `bitbucket.org/tclup/amr-dependency`.

[3]In using a two-layer BiLSTM, we follow Kiperwasser and Goldberg (2016).

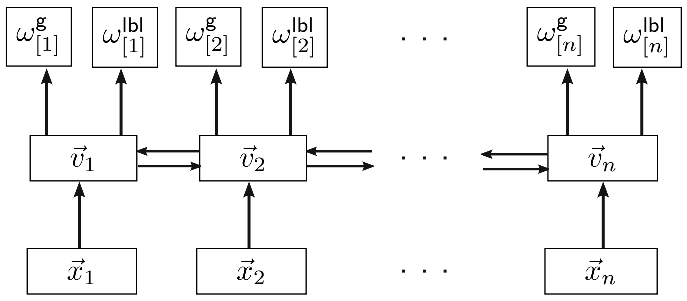

$\omega^{\mathsf{g}}_{[1]}$
$\omega^{\mathsf{lbl}}_{[1]}$
$\omega^{\mathsf{g}}_{[2]}$
$\omega^{\mathsf{lbl}}_{[2]}$
$\omega^{\mathsf{g}}_{[n]}$
$\omega^{\mathsf{lbl}}_{[n]}$
$\vec{v}_1$
$\vec{v}_2$
$\vec{v}_n$
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_n$

Figure 6.9: Architecture of the supertagger.

Supertagging Our supertagging model is a straightforward tagging model, and in its overall structure is similar to the approach of e.g. Lewis et al. (2016) for CCG supertagging.

Since we have finite lexicons for both the delexicalized constants and the lexical node labels (we use the constants observed in the training data to generate the lexicons), the domains of the supertag scoring functions $\omega^{\mathsf{g}}_{[i]}$ and $\omega^{\mathsf{lbl}}_{[i]}$ are finite. We can thus represent them as vectors, with each vector entry corresponding to one element in the domain, i.e. to a delexicalized constant for $\omega^{\mathsf{g}}_{[i]}$ and to a lexical node label for $\omega^{\mathsf{lbl}}_{[i]}$. To predict the scores, we add output layers on top of the BiLSTM as follows (see Figure 6.9):

$$\omega^{\mathsf{g}}_{[i]} = \log \mathrm{softmax}\left(A^{\mathsf{g}}\left(\vec{v}_i\right)\right)$$
$$\omega^{\mathsf{lbl}}_{[i]} = \log \mathrm{softmax}\left(A^{\mathsf{lbl}}\left(\vec{v}_i\right)\right)$$

where the output dimension of the affine transformation A^{g} is the size of the delexicalized constants lexicon, and the output dimension of A^{lbl} is the size of the lexical node label lexicon. We train the neural network using a cross-entropy loss function. This maximizes the likelihood of the supertags in the training data.

The supertagger reads inputs $x_i = (w_i, p_i, c_i)$, where w_i is an embedding of the word token, p_i an embedding of its POS tag,[4] and c_i is a character-based LSTM encoding of the word. We use pretrained GloVe embeddings (Pennington et al. (2014)) concatenated with learned embeddings for w_i, and learned embeddings for p_i.

[4]Adding POS tags to the training data has been shown to improve dependency parsing (Kiperwasser and Goldberg (2016)) as well as AMR parsing (van Noord and Bos (2017)). We obtain POS tags from Stanford CoreNLP (Manning et al. (2014)), using the Penn Treebank tag set.

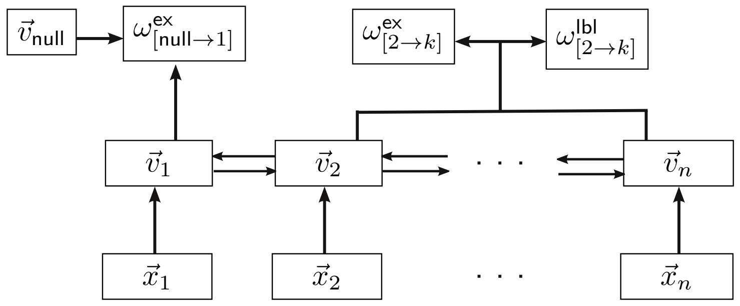

$\vec{v}_{\mathsf{null}}$
$\omega^{\mathsf{ex}}_{[\mathsf{null}\to 1]}$
$\omega^{\mathsf{ex}}_{[2\to k]}$
$\omega^{\mathsf{lbl}}_{[2\to k]}$
$\vec{v}_1$
$\vec{v}_2$
$\vec{v}_n$
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_n$

Figure 6.10: Architecture of the edge model.

Edge model For the edge model, we follow the approach of Kiperwasser and Goldberg (2016), except that we use a different loss function. We use the fact that every node in a dependency tree has at most one incoming edge, and train the model to score the correct incoming edge as high as possible. This model takes inputs $x_i = (w_i, p_i)$, where w_i and p_i are as above. The model is visualized in Figure 6.10.

We define the edge-existence and edge label scores as

$$\mathrm{MLP}_\theta(\vec{x}) = A_{\theta,1}\left(\mathrm{ReLU}\left(A_{\theta,2}\left(\vec{x}\right)\right)\right) \tag{6.4}$$

$$\omega^{\mathsf{ex}}_{[i\to k]} = \mathrm{MLP}_{\mathsf{ex}}(\vec{v}_i \circ \vec{v}_k) \tag{6.5}$$

$$\omega^{\mathsf{lbl}}_{[i\to k]} = \mathrm{MLP}_{\mathsf{lbl}}(\vec{v}_i \circ \vec{v}_k) \tag{6.6}$$

We further add a learned parameter $\vec{v}_{\mathsf{null}}$ to allow for the possibility that a word has no incoming edge. With Equation 6.5, we thus obtain scores $\omega^{\mathsf{ex}}_{[\mathsf{null}\to i]}$ for i having no incoming edge. While this score itself plays no further part in the model, during training it allows us to use the following loss function: To train the scores $\omega^{\mathsf{ex}}_{[k\to i]}$, we collect all scores for edges ending at the same node i into a vector $\omega^{\mathsf{ex}}_{[\bullet\to i]}$, including the score $\omega^{\mathsf{ex}}_{[\mathsf{null}\to i]}$ for no edge. At each position during training, exactly one entry of this vector is true. This allows us to minimize the cross-entropy loss for the vector $\mathrm{softmax}(\omega^{\mathsf{ex}}_{[\bullet\to k]})$, maximizing the likelihood of the gold edges. To train the labels $\omega^{\mathsf{lbl}}_{[i\to k]}(f)$, we simply minimize the cross-entropy loss of the actual edge labels f of the edges which are present in the gold AM dependency trees.

6.5 Decoding

Given learned estimates for the graph and edge scores, we now tackle the challenge of computing the best well-typed dependency tree t for the input string w, under the

$$\frac{s = \omega_{[i]}(c) \quad c \neq \bot \quad c \in \Sigma_0}{([i, i+1], i, \tau(c)) : s} \ \langle\text{Init}\rangle$$

$$\frac{([i,k], r, \tau) : s \quad s' = \omega_{[k]}(\bot)}{([i, k+1], r, \tau) : s + s'} \ \langle\text{Skip-R}\rangle$$

$$\frac{([i,k], r, \tau) : s \quad s' = \omega_{[i-1]}(\bot)}{([i-1, k], r, \tau) : s + s'} \ \langle\text{Skip-L}\rangle$$

$$\frac{\begin{array}{c}([i,j], r_1, \tau_1) : s_1 \quad ([j,k], r_2, \tau_2) : s_2 \\ \tau = f(\tau_1, \tau_2) \neq \mathsf{FAIL} \quad s = \omega_{[r_1 \to r_2]}(f) \quad f \in \Sigma_2\end{array}}{([i,k], r_1, \tau) : s_1 + s_2 + s} \ \text{Arc-R}$$

$$\frac{\begin{array}{c}([i,j], r_1, \tau_1) : s_1 \quad ([j,k], r_2, \tau_2) : s_2 \\ \tau = f(\tau_2, \tau_1) \neq \mathsf{FAIL} \quad s = \omega_{[r_2 \to r_1]}(f) \quad f \in \Sigma_2\end{array}}{([i,k], r_2, \tau) : s_1 + s_2 + s} \ \text{Arc-L}$$

Figure 6.11: Rules for the projective decoder.

score model (Equation (6.3)). The requirement that the term t must be well-typed is crucial to ensure that it can be evaluated to an AMR graph. But requiring well-typedness can also guide the decoding process, since it forces the resulting term to be globally consistent.

As shown in Appendix D, exact typed decoding is NP-complete. Thus, an exact algorithm is not practical. In this section, we develop an approximation algorithm for AM dependency parsing which assumes that the AM dependency tree is projective.

6.5.1 Projective decoder

The idea of the projective decoder is to build a well-typed dependency tree step by step, by combining adjacent substrings (i.e. spans). The decoder thus adds one operation at a time, type checking at each step. This adds a strong projectivity constraint discussed further below. The decoder uses dynamic programming to find the best tree under these restrictions.

The algorithm is shown in Figure 6.11 as a parsing schema, which derives items of the form $([i,k], r, \tau)$ with scores s. Such an item represents a well-typed derivation of the substring from i to k with head index r, and which evaluates to an as-graph of type τ. As we saw in Section 5.3, keeping track of the types is sufficient to ensure well-typedness. The head indices of the items allow us to apply edge scores $\omega_{[i \to j]}$ correctly, and with the spans we can track how much of the string we have covered, akin to plain CKY parsing with context free grammars.

The parsing schema consists of three types of rules. First, the Init rule generates at each position i an item for each constant c predicted for the token w_i, along with the score and type of that graph fragment. Second, the Skip rules allow us to extend a substring such that it covers tokens which do not correspond to a graph fragment (i.e., their supertag is $\perp$). Finally, given items for adjacent substrings $[i, j]$ and $[j, k]$, the Arc rules apply an operation f to combine the indexed AM terms for the two substrings, with Arc-R making the left-hand substring the head and the right-hand substring the argument or modifier, and Arc-L the other way around. We ensure that the result is well-typed by requiring that the types can be combined with f. After all possible items have been derived, we extract the best well-typed tree from the items with the highest score that span the whole sentence and have the empty type, i.e. items of the form $([1, n], r, [\,])$ for any r.

It is easy to see that the Arc-R and Arc-L rules have the highest parsing complexity, having parameters i, j, k, r_1, r_2, which yields a runtime of $O\left(n^5\right)$ (we interpret the complexity added by the parameters τ_1 and τ_2 as a grammar constant, since the number of possible types is fixed with a fixed lexicon). This parsing complexity is shared with other bilexical algorithms such as the Collins parser (Collins (1997)). It could potentially be improved to a complexity of $O(n^4)$ using the algorithm of Eisner and Satta (1999).

These rules allow us to associate each item $([i, k], r, \tau)$ with a partial dependency tree (covering all indices from i to $k - 1$) in the following way. For the Init rule, we simply obtain a constant paired with the index. For the Skip-R and Skip-L rules, we just pass the partial dependency tree of the argument along. When combining two partial dependency trees with an Arc rule for operation f, we simply take the union of the two partial trees (they do not overlap, since the spans are disjoint), and add an f-labeled edge between r_1 and r_2 (from r_1 to r_2 for Arc-R, and the other way for Arc-L). Since we build this dependency tree operation by operation, we ensure that there is in fact a well-typed term for the dependency tree.

IRTG perspective. In fact, from a certain perspective this algorithm builds a term rather than a dependency tree. We can see this by interpreting the given parsing schema as an IRTG. The idea[5] is to replace the i-th word w_i in the sentence with the number i itself. This allows us to anchor string constants at fixed positions, such that supertagger- and edge predictions can be interpreted correctly. So the sentence *The witch casts a spell* would be simply the sequence *1 2 3 4 5*. We then use the IRTG of Figure 6.12. For example, the Init$_{c,i}$ rule uses the literal string i,

[5]Inspired by a method originally used by Henning (2017)

grammar automaton	weight	h_{S}	h_{AM}
$\mathrm{Init}_{c,i} \to (i, \tau(c))$	$\omega_{[i]}(c)$	i	c
$\text{Skip-R}_k((r,\tau)) \to (r,\tau)$	$\omega_{[k]}(\perp)$	$*(x_1, k)$	x_1
$\text{Skip-L}_i((r,\tau)) \to (r,\tau)$	$\omega_{[i-1]}(\perp)$	$*(i-1, x_1)$	x_1
$\text{Arc-R}_{f,\tau_1,\tau_2}((r_1,\tau_1),(r_2,\tau_2)) \to (r_1,\tau)$ if $\tau = [\![f(\tau_1,\tau_2)]\!] \neq \mathsf{FAIL}$	$\omega_{[r_1 \to r_2]}(f)$	$*(x_1, x_2)$	$f(x_1, x_2)$
$\text{Arc-L}_{f,\tau_1,\tau_2}((r_1,\tau_1),(r_2,\tau_2)) \to (r_1,\tau)$ if $\tau = [\![f(\tau_2,\tau_1)]\!] \neq \mathsf{FAIL}$	$\omega_{[r_2 \to r_1]}(f)$	$*(x_1, x_2)$	$f(x_2, x_1)$

Figure 6.12: IRTG for the projective decoder; rules are defined for all $c \in \Sigma_0$, $i, j, k, r, r_1, r_2 = 1 \ldots, n$, $f \in \Sigma_2$ and types τ, τ_1, τ_2. Final states are $(r, [\,])$ for any r. The homomorphism h_{S} is to the string interpretation, where $*$ is simple concatenation; h_{AM} is the homomorphism to the AM interpretation. (Each label of the grammar occurs in only one row, so this table properly defines the homomorphisms).

thus being able to parse exactly the word position i. The rule then associates the supertag c with that position. We allow multiple such rules per word, so that the decoder can consider multiple supertags per word when searching for the best parse (see Section 6.8 for details). During parsing, this IRTG is combined with the string decomposition automaton we discussed back in Chapter 2. That decomposition automaton also uses pairs $[i, j]$ to represent string spans, and combining it with the IRTG rules in Figure 6.12 yields exactly the parsing schema in Figure 6.11. Thus, in a way, this algorithm builds a grammar custom made for a specific sentence, with weights obtained from the neural models. The nonterminals of this grammar are pairs (r, τ) of head indices and types, where the types ensure well-typedness and the indices allow us to apply the contextualized probabilities correctly.
Note that, while especially the formulation as an IRTG gives an AM *term* rather than a dependency tree, this term is arbitrarily chosen among the ones representing the highest scoring dependency tree. That is, while we technically directly obtain an AM term, we still rely on the work of Section 6.2 to know that we obtain the optimal AMR.

Projectivity constraint. In fact, the constraint here is a bit stronger than just projectivity. The parser can only build consecutive spans, and must perform the type checking at every step. For example, the dependency tree in Figure 6.13 is projective. However, the $\textsc{app}_{\mathsf{O2}}$ operation with argument *leave* must be performed before the $\textsc{app}_{\mathsf{O}}$ operation with argument *snake*, to resolve the unification at O before that slot is filled. This is not possible with this decoder, since it must combine *persuade*

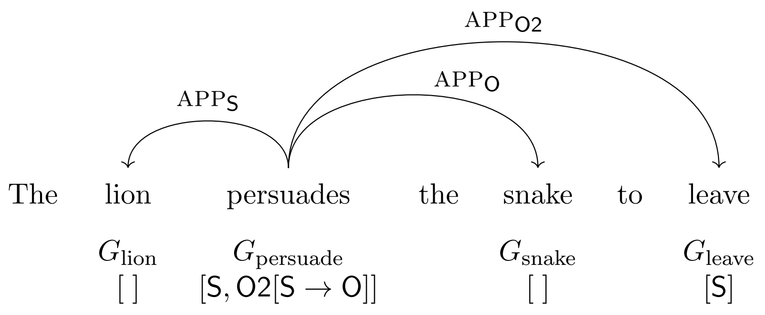


Figure 6.13: A dependency tree that the projective parser cannot obtain.

with *snake* first, and can only then combine it with *leave*. Implications and possible solutions to this issue are discussed in Section 6.9.

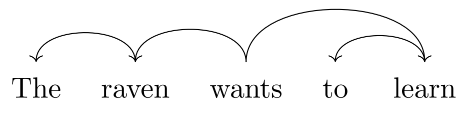
6.6 The related fixed tree model

This section briefly reviews an alternative approach to training the edge model, which we will call the *Kiperwasser & Goldberg edge model* (or K&G edge model) and an alternative approach to decoding, the *fixed tree decoder*. These approaches are also part of the original Groschwitz et al. (2018) paper, but were developed by Matthias Lindemann and are presented in more detail in his bachelor's thesis (Lindemann (2018)).

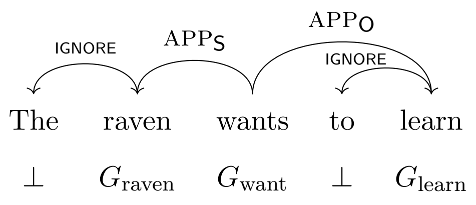
These approaches are closer to classic syntactic dependency parsing. They use additional edges in the AM dependency trees, labeled "IGNORE", such that all words in the sentence are covered. See Figure 6.14(b) for an example.

6.6.1 Alternate edge model

To train the edges, the K&G edge model follows the approach of Kiperwasser and Goldberg (2016) more closely. The difference to the edge model presented in Section 6.4 here is mostly that the K&G model uses a hinge loss, which compares the best predicted tree to the gold tree and computes a loss based on the differences found. This means that a tree needs to be predicted for every training iteration. The model developed by Lindemann for Groschwitz et al. (2018) uses the Chu-Liu-Edmonds algorithm (Chu (1965), Edmonds (1967)) for predicting these trees at training time, to account for the fact that the AM dependency trees are often non-projective (Kiperwasser and Goldberg (2016) predicts only projective trees).



(a) Step 1: Fix an unlabeled tree.



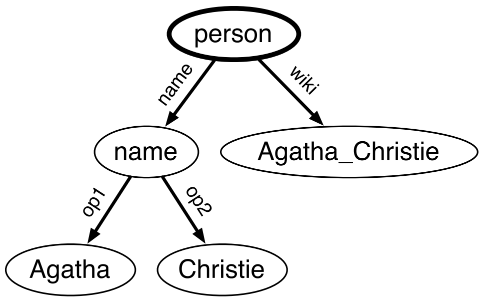
(b) Step 2: Typed decoding on the fixed tree structure.

Figure 6.14: The two steps of the fixed tree decoder.

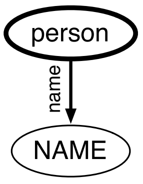
6.6.2 Alternate decoder

The fixed tree decoder has a two step process, see Figure 6.14. First, it uses the Chu-Liu-Edmonds algorithm to obtain an unlabeled dependency tree spanning the whole sentence. Here it is particularly handy that we factored the edge scores into scores $\omega^{\mathsf{ex}}_{[i\to k]}$ for edge existence, and scores $\omega^{\mathsf{lbl}}_{[i\to k]}$ for the edge labels. This first step computing the unlabeled tree only uses the edge existence scores. The decoder then fixes this unlabeled tree structure.

In a second step, the decoder then performs exact typed decoding on the fixed tree structure, exploring different combinations of edge labels and graph fragment supertags with dynamic programming. The fixed tree can have parts that correspond to words not contributing to the AMR, or parts that cannot participate in a high scoring and well-typed dependency tree. The second step can assign IGNORE labels and empty supertags, which will be ignored during evaluation, to 'prune' away these parts of the fixed tree. The fixed tree structure allows for efficient dynamic programming, and makes the typed decoding task tractable. The typed decoding algorithm on the fixed tree has a runtime of $O\left(n \cdot 2^d \cdot d\right)$, where n is the sentence length and d is the maximal arity of the nodes in the fixed tree. The algorithm returns the best well-typed dependency tree that follows the fixed tree structure. Thus, this decoder can obtain non-projective dependency trees and also the tree in Figure 6.13, at the cost of fixing the tree structure before checking type constraints.



(a) Before preprocessing



(b) After preprocessing

Figure 6.16: AMR representation of *Agatha Christie*, before and after preprocessing.

6.7 Data preparation

We have now fully established the AM dependency model in theory. To make the system work in practice, we need to add some pre- and post-processing steps concerning alignments, named entities, etc.

6.7.1 Training data

Alignments. We use a heuristic process to generate alignments between graph nodes and words. The process is described in Appendix B, and the code of the aligner is available open source at `bitbucket.org/tclup/alto`, in the class `de.saar.coli.amrtools.aligner.Aligner`. The aligner aligns every node in the graph to a single word[6], but multiple nodes can be aligned to the same word. For example in Figure 6.15, the aligner would align the *whistle-01* node to *whistles* and both the *bake-01* and *person* nodes to *baker*. Combined with the blob approach of attaching edges to nodes, as discussed in Section 5.5, these alignments partition the graph.

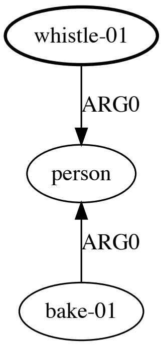


Figure 6.15: AMR for *The baker whistles.*

Names, dates and numbers. We use simple pre- and postprocessing steps to handle named entities, dates and numbers. In AMRs, named entities follow a pattern shown in Figure 6.16(a). Here the named entity is of type *person*, has a name edge to a *name* node whose children spell out the tokens of "Agatha Christie", and a link

[6]Or a span of words for named entities.

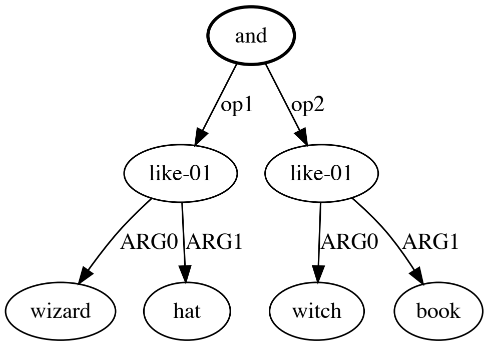


Figure 6.17: AMR for *The wizard likes hats and the witch books.*

to a wiki entry. Before training, we replace each *name* node, its children, and the corresponding span in the sentence with a special `NAME` token, and we completely remove wiki edges. In this example, this leaves us with only a *person* and a `NAME` node, see Figure 6.16(b). Further, we replace numbers and some date patterns with `NUMBER` and `DATE` tokens. On the training data this is straightforward, since names and dates are explicitly annotated in the AMR.

Obtaining the training data. We obtain AM terms as described in Chapter 5. The only difference is that the constants are slightly larger here: all nodes of one alignment are in the same constant, together with all their blob edges. We also employ the preferential source assignment model described in Section 5.6, i.e. preferring active over passive in the source assignments etc. Among the terms achieving the highest preference score, we then choose an arbitrary term t, and compute the dependency tree $\mathsf{dep}\,(t, \mathsf{ind})$ as described above, where the indexing function ind is given by the alignments. We did not find much variance in the dependency tree based on what term t we choose – compare this also to Conjecture 6.9. Note that we thus restricted the space of possible derivational structures for the AMR so much now, that we do not need statistical methods to find a consistent set of such structures.[7] We use Stanford CoreNLP (Manning et al. (2014)) for our POS tags to complete the training data.

Unusable graphs. Despite the edge deletion method to find AM terms for all the graphs, about 10% of the graphs (in the LDC2015E86 training set) cannot be turned into AM dependency trees. In a few cases because a graph was too large, but mostly because the aligner sometimes groups nodes together in a way that is not consistent

[7]That being said, some of the heuristic methods used here, like the aligner or the source assignment in the constants, could be replaced – and maybe improved – by statistical methods.

with our model. For example, the AMR in Figure 6.17 corresponding to the sentence

(1) The wizard likes$_i$ hats and the witch $\_\_i$ books

features ellipsis, i.e. the second occurrence of *likes* is omitted. Here then, both *like-01* nodes are aligned to the first occurrence of *likes* (the gap and reference notation '$\_\_i$' is not given in the corpus; we therefore cannot align one of the *like-01* nodes to the gap). In such a situation, we get more than one supertag for a single word position, which is not compatible with our model and we discard the sentence. This situation where multiple graph fragments get aligned to a single word are not only due to ellipsis, but also due to some AMR idiosyncrasies. The phenomenon occurs in about 7% of the graphs. We saw in Chapter 5 that about 2% of the graphs cannot be analyzed with the AM algebra using our heuristic constants. Together, this explains most of the unusable graphs.

We don't remove any graphs from the test data.

6.7.2 Evaluation data

For the evaluation data, we must first preprocess the sentence to replace names, dates and numbers with the corresponding tokens. Here, as opposed to the training data, we cannot look at the AMRs, but must use other methods. We detect dates and numbers with regular expressions, and names with Stanford CoreNLP.

Recall that we predict delexicalized graph fragments and their lexical labels separately. We find that just using the predicted lexical labels is suboptimal. Instead, we only use the supertagger's label prediction for words that were frequent in the training data (at least 10 times). For rarer words, we use simple heuristics, such as stemming and adding *-01* for verbs, explained in more detail in Appendix C.

For names, we look up name nodes with their children and wiki entries observed for the name string in the training data, and for unseen names use the literal tokens as the name, and no wiki entry. Similarly, we collect the type for each encountered name (e.g. *person* for "Agatha Christie"), and correct it in the output if the tagger made a different prediction. We recover dates and numbers straightforwardly.

6.8 Evaluation

We evaluate the AM dependency parser on the LDC2015E86 dataset, containing 16833 training instances, 1368 graphs in the development set and 1371 graphs in the test set, for a total of 19572 sentence-AMR pairs. We also use the LDC2017T10 dataset (also known as LDC2016E25), with 36521 training instances and the same development and test sets, for a total of 39260 sentence-AMR pairs. For brevity, I will refer to these datasets as "the 2015 dataset" and "the 2017 dataset" respectively.

We obtain AM dependency trees for the training and the development sets as described in Section 6.7. We then train the neural models for up to 100 epochs, choosing the epoch with highest supertagging accuracy on the development set. For evaluation, we predict supertag and edge scores on the test set and decode to obtain an AM dependency tree, which we evaluate to an AMR.

Implementation details. We trained the supertagging and edge scoring models of Section 6.4 separately; joint training (i.e. training the same BiLSTM module for both tasks) did not help significantly as seen below. We use 256 hidden dimensions in the BiLSTM and the MLPs, and employ dropout to prevent overfitting. Further details and hyperparameters can be found in Appendix E. We prune the set of supertags and edges we give to the decoder, for efficiency. The projective decoder is given 4 supertags per word, and only edges with a score over -4.6. We found this pruning to not hurt performance; supertags and edges with lower score seem to not participate in the best trees frequently.

6.8.1 Supertagger accuracy

We evaluate the accuracy of the neural supertagger on the development set of the 2015 dataset. Since the 'gold' supertags in the development set are also created heuristically (based on the aligner, heuristic source names and annotations, etc.), the presented accuracies are to be taken with a grain of salt.

Overall, the supertagger achieves an accuracy of 73%. The correct supertag is within the supertagger's 4 best predictions for 90% of the tokens, and within the 10 best for 95%.

Supertags that introduce grammatical reentrancies are predicted quite reliably, although they are relatively rare in the training data. The elementary as-graph for subject control verbs (see G_{want} in Figure 6.18(a)) accounts for only 0.8% of supertags in the training data, yet 58% of its occurrences in the development data are predicted correctly (84% in 4-best). The supertag for VP coordination (with type $\tau\mathsf{op1}[\mathsf{s}]$, $\mathsf{op2}[\mathsf{s}]$) makes up for 0.4% of the training data, but 74% of its occurrences are recognized correctly (92% in 4-best). Thus the prediction of informative types for individual words is feasible.

6.8.2 Baselines

We compare the AM dependency parser against two baselines.

Type-unaware fixed-tree baseline. We perform typed decoding, that is, our decoder ensures that the resulting AM dependency tree is well-typed. To investigate to what extent this is required, we consider a baseline which just predicts the best dependency tree, not taking types or projectivity into account (we use the fixed tree decoder of Section 6.6 for this). This leads to AM dependency trees which are not well-typed for 75% of the sentences (for the Smatch score evaluation below, we fall back to the largest well-typed subtree in these cases). Thus, the neural model does not automatically predict well-typed dependency trees, typed decoding is indeed necessary.

Graph decoder baseline. The AM dependency model has a close technical relation to the graph decoder parsers we described in Chapter 3, such as Flanigan et al. (2014) and Lyu and Titov (2018). Both parser formalisms predict graph fragments for each word, and predict edge scores between them. However, while we predict as-graphs and AM operations and decode into a tree, the graph decoders predict plain graphs and AMR edges, decoding into graphs.

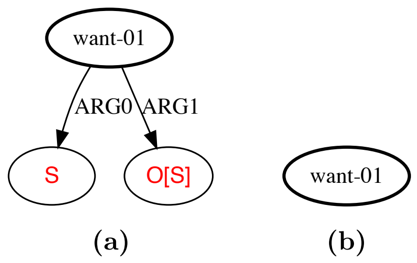


Figure 6.18: Graph fragments for *want* in (a) the AM dependency model and (b) a graph decoder.

We adapt our model to obtain a graph decoder baseline. Since a graph decoder predicts the edges which connect the constants separately from the constants, we remove outgoing blob-edges from our graph fragments, using the fragment in Figure 6.18(b) rather than the one in (a). That is, we also do not need our source name heuristics. Furthermore, we do not need to remove any edges in the training data to obtain sufficient coverage, and can use the original graphs.

We then train our neural model to directly predict AMR edges between these graphs. Since the assumption for the original edge model, that each word has only one incoming edge, does not apply here, we use a different loss function. Instead of using cross-entropy loss over the set of all incoming edges for a word, we use binary cross-entropy to predict the existence of each word separately.

When parsing a string, we choose the highest-scoring supertag for each word;

Model	**2015**	**2017**
AM dependency		
main model (projective decoder)	70.2±0.3	71.0±0.5
main edge model + fixed-tree decoder	69.4±0.6	70.2±0.5
K&G edge model + projective decoder	68.6±0.7	69.4±0.4
K&G edge model + fixed-tree decoder	69.6±0.4	70.2±0.2
Baselines		
fixed-tree (type-unaware)	26.0±0.6	27.9±0.6
graph decoder	66.1	66.2
Previous work		
Grammar-based		
Peng and Gildea (2016)	55	-
Neural sequence to sequence		
van Noord and Bos (2017)	68.5	71.0
Graph decoder		
JAMR Flanigan et al. (2016)	67	-
Foland and Martin (2017)	70.7	-
Lyu and Titov (2018)	**73.7**	**74.4**
Other		
Damonte et al. (2017)	64	-
Wang et al. (2015)	66.5	-

Table 6.1: LDC2015E86 & LDC2017T10 test set Smatch scores

there are only 628 different (delexicalized) supertags for the graph decoder, and 1-best supertagging accuracy is high at 88%. We then follow the graph decoding algorithm by selecting all edges whose score is over a threshold (we found -0.02 to be optimal; for log-probabilities as scores) and then adding edges until the graph is connected. Because we do not predict which node is the root of the AMR, we evaluated this model as if it always predicted the root correctly, overestimating its score slightly.

6.8.3 Results

Metric. We use Smatch score (Cai and Knight (2013)), the standard metric to measure similarity of AMRs, to evaluate our parser. Smatch score is based on triples of two forms. The first form is $\mathit{instance}(u, l)$ for a node u and a node label l, indicating that node u has label l. The second form is $\mathit{el}(u, v)$ for an edge label el and two nodes u and v, indicating that there is an edge with label el from u to v. Given a one-to-one map h between nodes of two graphs G and H, one can then compute precision, recall and f-score of these triples. For example, given a triple $\mathit{instance}(u, \mathit{love\text{-}01})$ in G, is there also a triple $\mathit{instance}(h(u), \mathit{love\text{-}01})$ in H? Or given a triple $\mathrm{ARG0}(u, v)$ in G, is there also a triple $\mathrm{ARG0}(h(u), h(v))$ in H? The Smatch score between two graphs G and H is then defined as the highest f-score of triples that can be obtained by any one-to-one map h between their nodes. Since the number of such maps is exponential in the size of the graphs, Cai and Knight (2013) introduce an approximate method to compute Smatch, which, while not exact, is shown to be reliable.

Results. Table 6.1 shows the Smatch scores of our model, compared to the baselines and a selection of previously published results. Our results are averages over 4 runs with 95% confidence intervals (graph decoder baselines are single runs). On the 2015 dataset, the AM dependency model as described here (results in first line) outperforms all previous work, with the exception of the Foland and Martin (2017) and Lyu and Titov (2018) models; on the 2017 set only Lyu and Titov (2018) outperforms the AM dependency model, the van Noord and Bos (2017) model ties.[8]

Switching out one or both of the edge model and decoder for the ones described in Section 6.6 does not improve results. Apparently, fixing the tree structure before typed decoding comes at a higher cost than only allowing projective trees.

As expected, the type-unaware baseline has low scores, due to its inability to produce well-typed trees. The fact that our models outperform the graph decoder baseline so clearly is an indication that they indeed gain some of their accuracy from the type information in the elementary as-graphs, confirming our hypothesis that an explicit model of the compositional structure of the AMR can help the parser learn an accurate model. While two graph decoder models (Foland and Martin (2017) and Lyu and Titov (2018)) outperform the AM dependency system, they use more elaborate training procedures and different pre- and post-processing. Thus, when comparing the AM dependency approach to the graph decoder approach, the comparison to our graph decoder baseline is more direct. Integrating the contributions made by Foland and Martin (2017), and especially Lyu and Titov (2018), with the AM dependency approach is a promising direction for improving parser performance further.

We clearly outperform the synchronous grammar approaches. Peng and Gildea (2016) obtain a smatch score of 55 on the 2015 dataset, 15 points below ours. The LDC2015E86 corpus was presumably not yet available for evaluation for Artzi et al. (2015); they only report a Smatch score of 66.3 on an earlier dataset (presumably LDC2014T10), where Smatch scores were in general a bit higher than on the newer datasets.

Table 6.2 analyzes the performance of our parser (AM) and the fixed-tree variant (FTD) in more detail, using the subtasks described in Damonte et al. (2017), and compares them to Wang's, Flanigan's, and Damonte's AMR parsers on the 2015 set,

[8]At the time of submission of Groschwitz et al. (2018), Lyu and Titov (2018) was not published yet, and the model tied for the best performing model on LDC2017T10.

	2015						2017			
Metric	W'15	F'16	D'17	L'18	AM	FTD	vN'17	L'18	AM	FTD
Smatch	67	67	64	**74**	70	70	71	**74**	71	70
Unlabeled	69	69	69	**76**	73	73	74	**77**	74	74
No WSD	64	68	65	**76**	71	70	72	**76**	72	70
Named Ent.	75	79	83	**85**	79	78	79	**86**	78	77
Wikification	0	75	64	**76**	71	72	65	**76**	71	71
Negations	18	45	48	**57**	52	52	**62**	58	57	55
Concepts	80	83	83	**86**	83	84	82	**86**	84	84
Reentrancies	41	42	41	**51**	46	44	**52**	**52**	49	46
SRL	60	60	56	**69**	63	61	66	**67**	64	62

Table 6.2: Details for the LDC2015E86 and LDC2017T10 test sets. AM is the AM dependency model of this thesis, FTD the alternate version of Section 6.6. W'15 is Wang et al. (2015), F'16 is Flanigan et al. (2016), D'17 is Damonte et al. (2017), L'18 is Lyu and Titov (2018) and vN'17 is van Noord and Bos (2017).

and van Noord and Bos (2017) for the 2017 dataset; we compare to Lyu and Titov (2018) on both datasets (Foland and Martin (2017) did not publish such results.) Mostly, the scores mirror overall parser performance. The still good scores we achieve on reentrancy identification, despite removing 60% of reentrant edges from the training data, indicates that we are particularly good at predicting the reentrancies we *can* model, such as control and coordination.

6.8.4 Structural evaluation

In this section, we examine where the gains of the main AM dependency model compared to the baseline come from, and also compare the AM model more closely to the current state of the art (Lyu and Titov (2018); abbreviated L'18 here). In particular, we will quantitatively evaluate performance on reentrancy structures that are created by control and coordination.

Metric		BL	L'18	AM
Label	R	81.1	83.2	78.9
	P	82.1	85.2	84.5
	F	81.6	84.2	81.6
Blob	R	71.0	78.7	77.6
	P	71.9	80.5	83.1
	F	71.5	79.6	80.3
Triangle	R	15.1	27.7	20.2
	P	17.4	36.6	43.4
	F	16.2	31.5	27.5
Conjunction	R	7.0	38.1	28.1
	P	43.2	68.9	67.3
	F	12.1	49.0	39.7

Table 6.3: Results of structural evaluation on the LDC2017T10 test set. BL is baseline, L'18 is Lyu and Titov (2018) and AM is the main AM dependency model.

Smatch score relies on a node mapping between the graphs, which is influenced by several properties of the graphs, in particular by both structural properties and node labels. Since many of the more detailed metrics shown in

Table 6.2 also rely on Smatch score, this applies there as well (a notable exception is the Negations metric). In this section, we will disentangle node labels and structural properties by using metrics that do not rely on a node mapping.

The results of the evaluation on the 2017 test set are shown in Table 6.3. All metrics are recall (R), precision (P) and f-score (F) on multisets of local phenomena.

The first metric **Label** simply compares the multiset of all node labels in the gold graph against the respective multiset of the predicted graph; i.e. this is a 'bag of node labels' metric. The AM parser does not improve over the baseline here, which means its improvements are entirely structural, i.e. in the way the nodes are connected. In contrast, L'18 performs noticeably stronger in this metric, showcasing improvements orthogonal to the ones developed here. The AM parser has relatively low recall and high precision, meaning that it predicts smaller graphs on average; this may be due to the parser being more restrictive.

The second metric **Blobs** considers the multiset of blobs in the graph. Recall that the blob of a node is the collection of incident edges that 'belong' to it, such as outgoing argument edges of a predicate or incoming mod edges of a modifier. We can thus model a blob as a multiset of edge labels, that is, we describe the blob of a node as the multiset of the edge labels of the node's blob edges. For example, the *try-01* node in the AMR in Figure 6.19 has the blob $\{\text{ARG0}_1, \text{ARG1}_1\}$, where the indices indicate multiplicity. The AMR then has the blob multiset

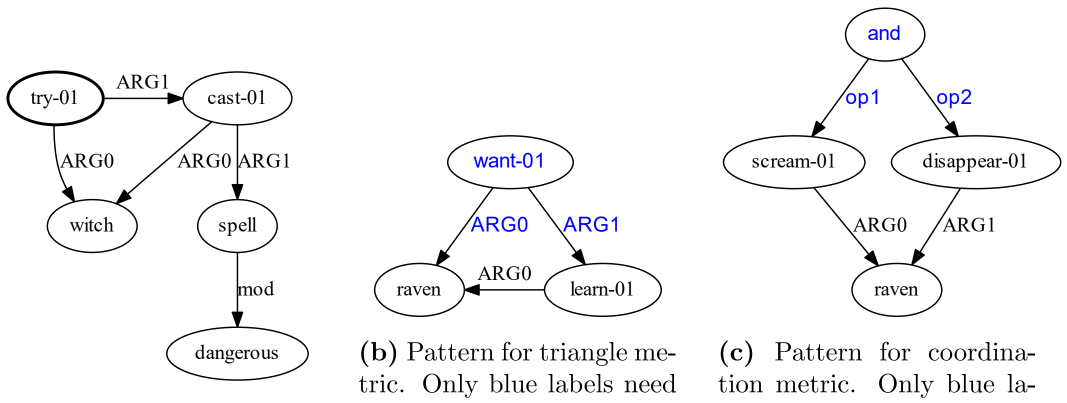
$$\{\{\text{ARG0}_1, \text{ARG1}_1\}_2\,, \{\text{mod}_1\}_1\,, \emptyset_2\}\,.$$

In this metric, the AM dependency parser improves thoroughly on the baseline and even outperforms L'18. This is most likely because the AM parser predicts supertags that already contain meaningful combinations of edges in the blobs.

The third **Triangle** metric measures the ability to predict triangles of edges where two edges belong to the blob of one of the nodes. An example is shown in Figure 6.19(b), where both blue edges are in the *want-01* blob. Such short range reentrancies are captured by one of the source annotation heuristics and are often created by compositional phenomena such as control. When such a pattern is caused by control, the node label indicated blue in Figure 6.19(b) is the control verb. Table 6.4 shows the top five node labels in the 2017 test set that occur as the blue node in this pattern. The node labels *want-01* and *try-01* indicate

Label	Count
feel-01	21
want-01	20
have-org-role-91	15
say-01	15
try-01	14

Table 6.4: Top 'blue' nodel labels of the Triangle pattern on the LDC2017T10 test set.



(a) An AMR.

(b) Pattern for triangle metric. Only blue labels need to match.

(c) Pattern for coordination metric. Only blue labels need to match.

Figure 6.19: An AMR and evaluation metric patterns

control, and a triangle with *feel-01* can occur in constructions such as *I feel happy*. Triangles for *have-org-role-91* and *say-01* may be due to coreference.

The Triangle metric then compares the multisets of such triangle structures for the gold and the predicted graph. For two triangle structures to count as equal in the Triangle metric, only the blue edge and node labels need to match. The idea behind this decision is to decouple the structural evaluation as much as possible from node and edge label prediction, while still capturing the essence of a control structure when one occurs. For the Triangle metric, the AM parser again improves strongly over the baseline and is close to L'18 in f-score. Accuracy for the AM parser is particularly high while recall is low. This may be because the AM algebra captures some phenomena that cause triangles well (such as control), and others less well (such as coreference).

Conjunction patterns are matched by the eponymous **Conjunction** metric. Like the triangle metric it matches graph patterns, namely those conjunction patterns where the conjuncts have a common argument, like the conjunction shown in Figure 6.19(c). Again, only the blue node labels and the graph structure need to match. That is, the conjunction label itself needs to match, and the number of conjuncts, as well the number of joint arguments of the conjuncts. This metric is where the AM algebra has the largest improvements over the baseline, although L'18 performs even stronger.

Overall we find that the AM algebra strongly improves over the graph decoder baseline in structural terms, including two important reentrancy patterns, but not in terms of node labels. The Label metric is also one where the difference to Lyu and Titov (2018) is especially pronounced. These results, in particular those of the Blob metric, are evidence for the hypothesis that the compositional approach of the

AM parser has structural benefits over the graph decoder method.

6.8.5 Model variations

Table 6.5 shows results for different neural model variations of the main AM dependency parser on the development set of LDC2015E86. First we test whether joint training, i.e. training one BiLSTM as input to both the supertagger and the edge model, makes a difference. Recall that the supertagger presented above uses as word vectors a concatenation both (fixed) pretrained word embeddings and learned word embeddings, as well as POS tag embeddings and an LSTM-based character-by-character encoding of the word. The edge model uses the same vectors, except the character-by-character encoding.

Model variant	Smatch
non-joint full	71.3
joint full	71.4
- char	71.2
- POS	70.5
- learned	70.2
512	71.7
128	69.4

Table 6.5: Scores of model variations on the LDC2015E86 development set

Since the joint model (*joint-full* in Table 6.5) uses one fixed configuration for both supertag and edge predictions, we compare it to a non-joint model where the edge model also uses the character-by-character encoding, to get a fair comparison (*non-joint full* in Table 6.5). The two models vary only by 0.1 Smatch score, not a meaningful difference given the error margins we saw in Table 6.1.

For the further ablation studies, we use the joint model since it was more convenient to use. We first do some ablation studies. Removing the character-by-character encoding from the input vectors costs about 0.2 points of Smatch score ("- char" in the table), although this may not be significant. Removing also the part of speech encodings ("- POS") costs 0.7 points, and removing the learned word embeddings as well, leaving only the fixed pretrained embeddings, costs another 0.3 points ("- learned"). In total, using only the pretrained embeddings yields a Smatch score of 70.2 on the development set, costing more than a full point in performance.

We also experiment with the number of hidden states. Increasing the number of hidden states from 256 to 512 increases the development Smatch score to 71.7, a minor 0.3 increase (in the range of random variation). Decreasing the number of states to 128 costs 2 full points of Smatch score. That is, the number of hidden states we use is necessary for good performance, but increasing it further (which increases training runtimes) does not give much additional benefit.

Further, we evaluated the effect of the source name preferences described in Section 5.5.2. This performed this evaluation with the full neural main model on the LDC2017T10 test set, achieving a Smatch score of 68.0 without the source name preferences, compared to the 71.0 with the preferences. This illustrates how making the training data more consistent can have a considerable impact on performance.

6.9 Discussion

We have now seen the AM dependency parser and its empirical evaluation. In this section, we look at two more examples, discuss the impact of compositionality on the parser, and take stock of how we addressed the challenges put forth in the introduction.

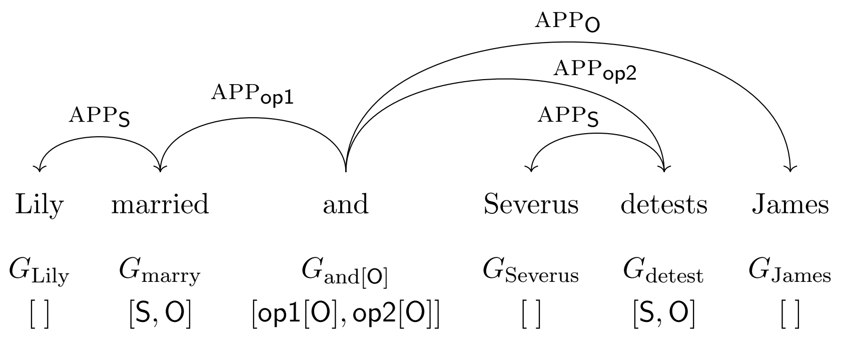
6.9.1 Examples

Throughout this chapter we have already seen many examples of AM dependency trees. We also analyzed properties of the AM algebra in Section 5.4, most of the examples discussed there translate directly to dependency trees. Further, we saw that ellipsis and coreference are clear limitations of the current model. There are however two phenomena we described in Chapter 3 that we did not discuss yet: right node raising and wh-movement. The two phenomena highlight strengths of the dependency approach, and a limitation of the projective decoder.

Right node raising Recall right node raising, as in

(2) Lily married and Severus detests James

where the segments *Lily married* and *Severus detests* are coordinated, even though they are not usually considered constituents. However, the AM dependency parser considers only the semantic types, and no presupposed notion of syntax, to determine which dependency trees are allowed. Thus, the dependency tree

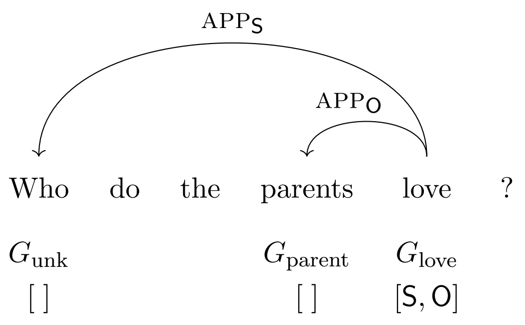


is perfectly acceptable for the AM dependency parser, and provides a simple solution to the right node raising problem.

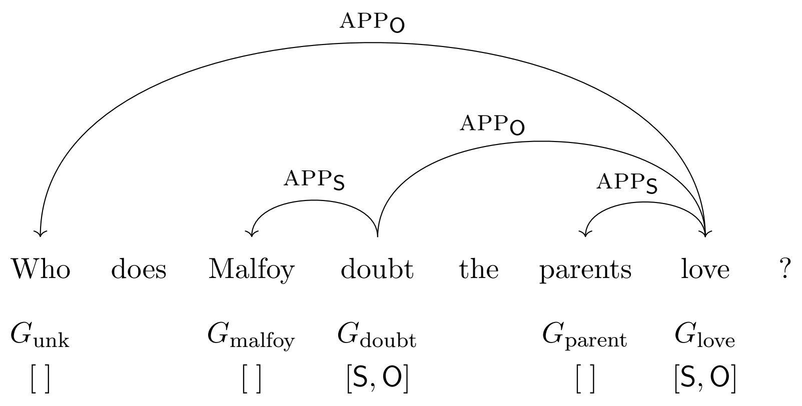
Wh-movement The problem with wh-movement is the unusual position of the wh-word indicated by the gap $\_\_i$ in

(3) Who$_i$ do the parents love $\_\_i$?

Again, this no problem in the AM dependency model as illustrated by this dependency tree:



Now consider the following dependency tree for long-distance wh-movement:



This is a non-projective tree: the long distance APP_O edge from *love* to *Who* crosses from one side of the root (i.e. origin) of the dependency tree to the other. This is no problem for the AM dependency model as a whole, but the approximate projective decoder cannot obtain this: it would need to combine *love* and *Who* before combining *love* further as the object of *doubt*, which would cover a non-consecutive part of the sentence. Earlier in Section 6.5, we saw a similar problem occurring with object control.

We can look to mildly context-sensitive grammars for solutions that relax the projectivity constraint while keeping computational complexity manageable. Many mildly context-sensitive formalisms, for example CCG, allow an unfilled argument slot to be 'passed along' via function composition and be filled only later. That is, *Malfoy*, *doubt*, *parents* and *love* would combine first, and only then would *Who* be added as the object of *love*. Tree Adjoining Grammar (TAG, Joshi and Schabes (1997)) uses a different approach during parsing. Here, non-consecutive sentence segments, specifically pairs of spans, are allowed as states in the parsing process. In fact, Koller and Kuhlmann (2012) introduce a string algebra that models TAG, which we could plug into the IRTG perspective of the projective decoder.

In conclusion, the AM dependency model handles many phenomena with ease that often pose a challenge to compositional models. Projectivity constraints are an issue, but could be relaxed in the future. To obtain this flexibility, the AM dependency model gives up on the tight connection to traditional syntax that is characteristic of compositional models. We discuss this in the following section.

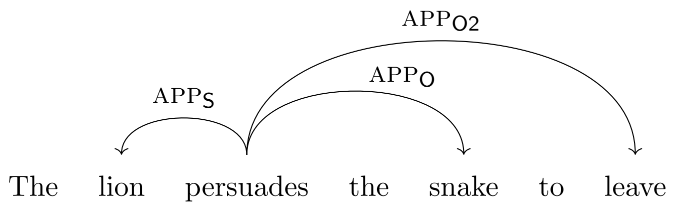
6.9.2 Compositionality

In Section 3.2, we established the principle of compositionality as

> The meaning of a complex expression is a function of the meanings of its parts and the mode of composition by which it has been obtained from these parts.

Here, we understand the "mode of composition" to refer to the syntax of a sentence. Also in Section 3.2, we said that we call a parser compositional if it defines a syntax and constructs the meaning representation along the syntax structure. So, is the model discussed here compositional?

Dependency model. For the (somewhat hypothetical) dependency model with exact typed decoding, as described in Section 6.3, the answer is: it depends. If we project the edges of an AM dependency tree onto the sentence, such as in



(6.7)

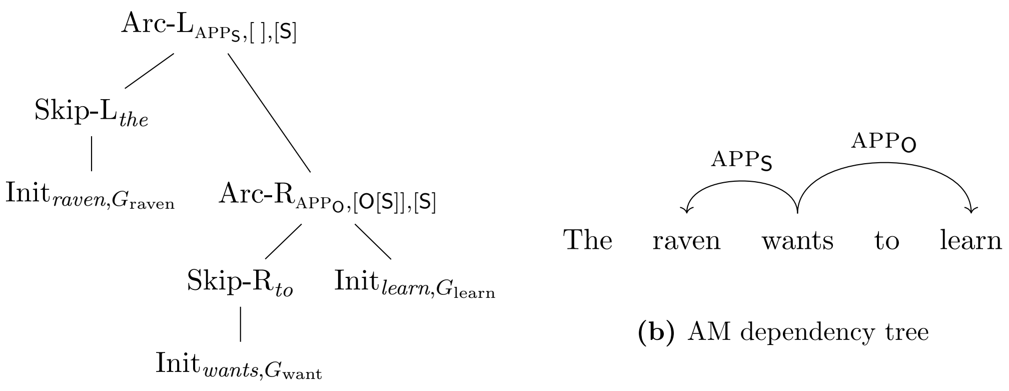
Rule	h_{S}	h_{AM}
$\text{Init}_{w,c} \to \tau\,(c)$	w	c
$\text{Skip-R}_w\,(\tau) \to \tau$	$*\,(x_1, w)$	x_1
$\text{Skip-L}_w\,(\tau) \to \tau$	$*\,(w, x_1)$	x_1
$\text{Arc-R}_{f,\tau_1,\tau_2}\,(\tau_1,\tau_2) \to \tau$ if $\tau = [\![f\,(\tau_1,\tau_2)]\!] \neq \mathsf{FAIL}$	$*\,(x_1, x_2)$	$f\,(x_1, x_2)$
$\text{Arc-L}_{f,\tau_1,\tau_2}\,(\tau_1,\tau_2) \to \tau$ if $\tau = [\![f\,(\tau_2,\tau_1)]\!] \neq \mathsf{FAIL}$	$*\,(x_1, x_2)$	$f\,(x_2, x_1)$

Figure 6.20: A 'global' IRTG inspired by the projective decoder. Rules are defined for all observed as-graph fragments c, all observed words w, all AM operations $f \in \Sigma_2$ and possible types τ, τ_1, τ_2. The final state is the empty type.

then this defines a dependency structure on the sentence. We can interpret this as a form of "syntax" on the sentence. This "syntax" then, combined with the meanings (i.e. the as-graphs) of the single words, determines an AM dependency tree, and as we have seen in Theorem 6.8, this fully determines the resulting AMR, which in this context is the 'meaning' of the sentence.
However, calling this dependency structure a syntax may be a bit of a stretch. The principle of compositionality refers to "the mode of composition by which [the complex expression] has been obtained", and quite clearly, the dependency structure in (6.7) does not fully describe how the sentence has been obtained. To obtain a more traditional syntax, one could add the IGNORE edges of Section 6.6.2 to cover all words in the sentence, and use a dependency grammar as a syntactic model. However, this might add some of the harder restrictions back in that we explicitly tried to avoid.

Projective decoder. If we take the projective decoder into account, we can obtain another, different perspective on compositionality. As we saw in Section 6.5, we can interpret the projective decoder for a specific sentence as an IRTG (see Figure 6.12). We can interpret that 'local' IRTG as a version of a 'global' IRTG, shown in Figure 6.20, with contextualized probabilities and a pruned rule set. The (hypothetical) process is as follows: The 'global' IRTG in Figure 6.20 uses AM types as nonterminals, and contains the following types of rules:

- $\text{Init}_{w,c}$ that maps any word w to any graph fragment c.
- Skip-R_w and Skip-L_w that skip any word w, leaving the as-graph representation unchanged.



Arc-L$_{\text{APP}_S,[\,],[S]}$
Skip-L$_{the}$
Init$_{raven,G_{\text{raven}}}$
Arc-R$_{\text{APP}_O,[O[S]],[S]}$
Skip-R$_{to}$
Init$_{learn,G_{\text{learn}}}$
Init$_{wants,G_{\text{want}}}$
APP$_S$
APP$_O$
The
raven
wants
to
learn

(a) IRTG term

(b) AM dependency tree

Figure 6.21: A term over the IRTG in Figure 6.20 for the sentence *The raven wants to learn*, and a matching AM dependency tree.

- Arc-R$_{f,\tau_1,\tau_2}$ and Arc-L$_{f,\tau_1,\tau_2}$ that apply an operation f to two arguments of types τ_1 and τ_2.

For example, the sentence

(4) The raven wants to learn

can be described with the term in Figure 6.21(a); the term corresponds to the AM dependency tree in (b). To obtain the 'local' IRTG of Section 6.5, we add the probabilities as given by the dependency model, replace words with indices and add nonterminals as indicated in Section 6.5, to be able to score the rules properly in the local context. Replacing the words with indices is only a trick to get the right contextualized probabilities. We then prune the IRTG, by only keeping the few best constants for each word position (i.e. only keeping a small set of supertags) and only keeping edge rules with probabilities over some threshold.

From this perspective then, the model is fully compositional, with a probability model that factors in a specific way such that all terms corresponding to the same dependency tree have the same total score. Since we designed the application and modification operation manually, from some perspective we actually 'handwrote' large parts of this grammar.

An interesting difference to a more classical synchronous grammar approach is that this grammar here is very permissive. The nonterminals only track semantic well-typedness, and otherwise operations can combine freely. To some extent, this mirrors ideas present in e.g. Hall et al. (2014), that less information is encoded in the nonterminals, and more in the probabilities. This makes the grammar more flexible, allowing it to consider more options and putting more focus on the probability model to disambiguate between them.

Conclusion Technically compositional or not, the AM dependency model uses many ideas present in more traditionally compositional models. These ideas are encoded in the AM terms. Solutions to remaining problems will quite likely also draw from this tradition, see e.g. the possible solutions to the projectivity constraint discussed above. Thus, the AM dependency model clearly lies in the tradition of compositionality.

6.9.3 Comparison to other AMR parsers

Let us compare the AM dependency approach to other high performing approaches to AMR parsing, namely the neural seq2seq approach (e.g. van Noord and Bos (2017)) and the graph decoder approach (e.g. Lyu and Titov (2018)).

Data hunger. The neural seq2seq approach of van Noord and Bos (2017) relies heavily on silver data. This may be because a seq2seq model must not only learn to generate *good* graphs, but must also learn to generate a string that actually represents a graph. It may be that pre-training on the silver data allows the seq2seq model to learn more basic aspects of the task (that other approaches have already built in), and then refine its predictions on the gold data. Since existing parsers (that are at least reasonably good) are required to generate silver data, this approach seems less appealing for newly emerging semantic formalisms where no parser has been developed yet.

Both the graph decoder models and the AM dependency approach do well without additional silver data. In future research, it would be interesting to see whether the additional structure in the AM dependency trees helps to generalize from limited training data. Experiments to test this hypothesis could include parsing on particularly small sets of training data, seeing whether additional silver data helps one approach more than the other, or evaluating on test data of a different domain. Possible data for such experiments could include manually restricted training sets of the AMR corpora used here, the Little Prince AMR corpus, the Bio AMR corpus, and the silver data used by van Noord and Bos (2017).

Incorporating linguistic principles. Graph decoders select the edges for a graph largely independently. While we have discussed in Section 3.4 some ways in which already predicted edges can influence following predictions (such as each node being restricted to at most one of each ARGx edge), different constellations of edges are not inherently favored or unfavored. In contrast, AM dependency parsing groups edges in meaningful supertags and encode reentrancies in compositionally principled ways. This yields measurable improvements in graph structure, as we saw in Section 6.8.4.

In their seq2seq model, van Noord and Bos (2017) make several structural decisions that treat each AMR as not just a sequence of characters, such as using principled linearizations of the graphs and treating edges as a single supercharacter rather than a sequence of independent characters. However, it is not clear how to incorporate deeper compositional principles in a seq2seq approach. Since there can also be advantages to not adding linguistic principles to a model – such as lower development time – the different approaches may well be complementary, with the choice depending on the requirements and restrictions of each application.

6.9.4 Taking stock

In the introduction of this thesis, I listed four challenges to neural- and grammar-based parsers. To recap them:

Challenge 1: Models without linguistic structure are data hungry.

Challenge 2: It is not obvious how to add linguistic principles to neural networks.

Challenge 3: Synchronous grammars face robustness issues.

Challenge 4: Creating structural training data is a highly ambiguous process.

So, how does the AM dependency parser address these challenges?

Combining neural networks with linguistic principles (Challenges 1&2). The AM dependency parser is based on the general linguistic principles of application, modification and unification; as we discussed in Chapter 3, these principles are present in many compositional formalisms for semantic construction, such as LFG or CCG. By predicting operations over the AM algebra, we do not add the linguistic structure inside the neural networks, but use the neural networks to predict the structure.

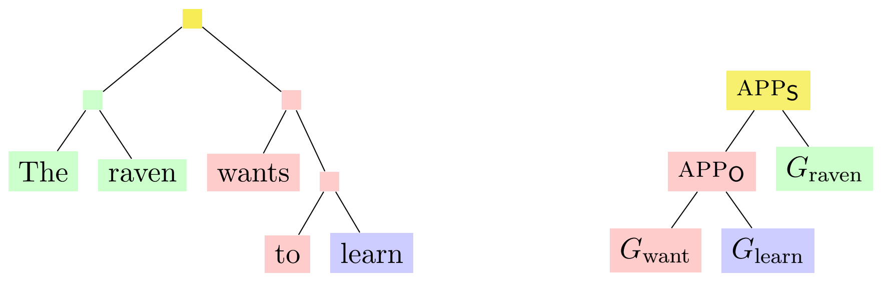
As a result, we outperform the sequence-to-sequence approach of van Noord and Bos (2017) on the LDC2015E86 dataset, and tie on the larger LDC2017T10 dataset. However, van Noord and Bos (2017) use additional ‘silver’ training data created by other parsers. This underlines that our more structured approach requires less data to work effectively than van Noord and Bos (2017). While some graph decoder parsers (Foland and Martin (2017), Lyu and Titov (2018)) outperform our parser, they use more complex neural architectures, as well as different pre- and post-processing. These can have big impacts on parser performance. The fact that we outperformed our graph decoder baseline, that uses a near identical neural architecture and the same pre- and post-processing steps, indicates strongly that guiding the parser with linguistic principles does indeed help. Integrating the advanced training regimes of in particular Lyu and Titov (2018) into the AM dependency model is a promising direction for further parser improvements.

In the AM dependency parser, the connection between linguistic principles and the parsing process is more explicit and overt, compared to purely neural models or the graph decoders. This is a double edged sword. On the one hand, it allows us to identify problems and possible solutions more explicitly. We saw in Chapter 5 that coreference is an unsolved problem, and in this chapter that we cannot model ellipsis. These open issues give specific avenues for future work, and much of the specifics of the final model presented here originated from addressing similar overt issues.

The downside is that with the added structure of the AM algebra, we *have to* address these issues specifically – otherwise, we cannot model them at all. For example, in Section 5.5.3 we saw some phenomena that our heuristics to extract constants and source annotations do not cover. To extend the heuristics manually, until all phenomena in the corpus are covered, would imply addressing the Zipfian tail of language with manual labor, an ineffective approach. Using statistical methods to replace or extend the heuristics used in this thesis could be promising moving forward.

We thus have the options to address overt issues with custom tailored, linguistically inspired solutions, and/or to add more statistical models in the mix, to address the Zipfian tail. Which issues to address directly, and which to address with broader statistical methods, is a delicate balance. The evaluation in this chapter shows that so far, the AM algebra strikes that balance successfully – the simple heuristics we use are effective.

Robustness (Challenge 3). While the AM dependency model uses the same principles of argument application, modification and unification as other grammar-based approaches to synchronous parsing, such as LFG or CCG, there is a key difference here. In the AM dependency model, we use no syntactic categories, and have no formal restrictions on word order (besides the projectivity constraints of the approximate decoder). While the type system contains information about available argument slots, a notion often encoded in syntactic categories, we do not make more fine-grained distinctions between e.g. noun phrases and clauses, i.e. the syntactic



(a) (Simplified) syntax structure. **(b)** Term over the AM algebra.

Figure 6.22: Syntax structure and AM term for *The raven wants to learn*, with colors indicating matching parts.

types NP and S. Instead, we rely on the neural model to learn this kind of information implicitly, as is needed. This is indeed effective, since the bidirectional recurrent neural networks provide our decoder with accurate scores that take context into account. Thus, the AM dependency model avoids many of the hard constraints of synchronous grammars, without giving up on the linguistic principles of application etc. This makes the model much more flexible and robust.

There is another insight to be found here, which is a bit more subtle. While we have the projectivity constraint at decoding time, the dependency trees we use in the training set are unconstrained. This solves a crucial robustness issue that synchronous grammars face during training.

Recall that synchronous grammars assume that for any given sentence, there is a syntax term and a semantic term (such as the ones shown in Figure 6.22), that we can cut into corresponding parts (here indicated by colors) that are consistent with the tree structures, loosely that the ways the trees are cut up are congruent.[9] Thus, at training time, one needs to find a syntax and a semantic term, a segmentation for each, and a way to match segments of the two terms with each other, such that the parent-child relations of the matched segments are congruent in both terms. Optimally, the matching of segments in both terms should be also consistent with a given set of heuristic alignments between the AMR and the string. These are strong constraints that are difficult to satisfy. For example, Peng et al. (2015) note that their induction algorithm is particularly sensitive to the heuristic alignments.

Preliminary experiments on inducing a synchronous IRTG with the AM algebra, conducted by my colleagues Christoph Teichmann and Antoine Venant (personal communication, 2018), indicate that when assuming a fixed syntax structure, fixed

[9]What exactly *congruent* means here depends on the grammar formalism, for example for IRTG it means that there is a derivation tree, such that both terms are homomorphic images of the derivation tree.

alignments and fixed AM term, a congruent matching can barely be ever found.[10] To a large extent, this is because none of the terms or alignments are hand-annotated or fully correct; for example one misaligned node in the graph can completely break the congruence of the terms.

When, instead of fixing a single pair of terms and their alignments, we sample from all possible terms and alignments, the same sort of complexity problems that we observed for the HR terms in Chapter 4 occurred again, such that the sampling process could not find consistent patterns. This time, the too large sampling space comes not from the number of semantic terms, but rather from the combination of the different possible syntactic and semantic terms, alignments, and different ways to segment the terms. The experiments were inconclusive on whether it is possible to find a 'sweet spot', i.e. a somewhat restricted sampling space where we still *can* find congruent terms with matching alignments, while avoiding the complexity problems. In any case, this is a non-trivial task.

The AM dependency model completely sidesteps this issue. Since the AM dependency trees at training time do not have projectivity constraints, or any constraints concerning word order, we can simply project them onto the sentence according to the given alignments, and that's it.

Ambiguity in structural training data (Challenge 4) In Chapter 4 we obtained billions of HR terms even for a single very small AMR. Using statistical sampling methods to obtain a consistent set of terms for training a parser was infeasible. The AM algebra drastically reduced this ambiguity when selecting terms for a given AMR, even more so when adding a simple preferential heuristic for source name choices. AM dependency trees reduced this ambiguity even further: Conjecture 6.9 proposes that we now only need to fix the constants to obtain a unique dependency tree, and indeed, using heuristics for the constants and then just picking an arbitrary corresponding AM dependency tree gives us a consistent set of trees for training. With this drastic reduction in ambiguity, we do not even have to rely on statistical methods anymore.

While we no longer rely on statistical methods to select an AM dependency tree, we do rely on a series of heuristics for alignments, and for the shape, sources and annotations of the constants. Replacing these heuristics with statistical methods might help to improve the model, for example, Lyu and Titov (2018) saw significant performance increases by using an advanced technique for learning alignments.

[10] At least not when assuming reasonably small segments. One can always choose one big segment for each term, and match the two, but such a rule will of course not generalize, it just learns the sentence by heart.

An important note here is that we mostly removed undesirable terms or spurious information within them. The AM algebra brings the HR terms into a normal form, and disallows terms that do not fit into the application/modification/unification schema. While this means we can no longer some phenomena, the large number of spurious terms we no longer have to consider is worth it. And as we saw, we only need to remove 4% of edges to achieve nearly full coverage. The AM dependency trees then essentially just summarize multiple terms that evaluate to the same AMR anyway. Thus, where the HR algebra is a very powerful algebra that can build any graph in many different ways, the AM dependency trees seem to be exactly powerful enough to build a semantic parser.

6.10 Conclusion.

In this chapter, we saw that AM dependency trees encode the essence of AM terms, and that we can use them to build an AMR parser with strong performance. The parser combines the strengths of neural and compositional approaches, addressing all challenges we established in the introduction.

Future work. The parser presented in this section uses a simple, straightforward neural model. There are many related neural models that have been used in other work, and could be adapted for AM dependency parsing. For example, Dozat and Manning (2017) use biaffine attention for dependency parsing, and Lyu and Titov (2018) use bilinear edge scores and a multi-pass approach for their edge model. Experimenting with different supertagging models, such as the one of Lewis et al. (2016), could also be interesting.

Furthermore, the transparency of the system allows us to clearly identify remaining limitations, such as coreference and ellipsis, providing directions for future work. The next chapter will discuss this in more detail.

7 Conclusion

This thesis presented the AM dependency parser, a semantic parser for AMR that combines neural networks with ideas from compositional semantic parsing in a new, flexible dependency structure. It effectively reduces the complex graph parsing task to a simple dependency parsing task, allowing us to use straightforward neural methods for supertag and edge prediction. The AM dependency parser relies less on hard syntactic constraints than more traditional compositional approaches, instead using the contextualized scores of bidirectional recurrent neural networks combined with a semantic type system to guide it. This simplifies it and increases its robustness compared to synchronous grammar based approaches.

At the same time, we tackled the issue of ambiguity when creating structural, compositional training data from a corpus. We started with the decomposition automata of the HR algebra that gave us billions of different terms and automata with millions of rules, even for small AMRs. This made a sampling approach to obtain training data unfeasible. We analyzed the reasons for this ambiguity in creating terms, and developed the AM algebra with higher level operations for semantic construction. This led to a drastic reduction of possible terms for each graph, while keeping high quality terms available. The AM dependency trees that underspecify AM terms helped further with this issue, to the point where we obtained a consistent set of training data even without sampling.

We thus obtained a semantic parser that combines current, powerful neural methods with a gentle guidance based on traditional compositional approaches, solving conceptual and technical challenges along the way.

7.1 Future work

Covering more phenomena. The transparency of the AM dependency model reveals clear opportunities for improvement. Current limitations include the phenomena of coreference and ellipsis. While these are challenging phenomena and solutions are not obvious, the problems are clearly defined, making them readily available targets for future research.

Learned decomposition decisions. We also saw that the heuristics we use in some steps to process the training data still have some flaws (e.g. the source annotation heuristic). While they could be improved manually, this might be tedious and ineffective. Statistical methods for these tasks could be a helpful addition in the future. For example, techniques like Expectation-Maximization (EM) are designed to find the latent structures that best explain the training corpus; here, significant runtime issues might arise. Reinforcement learning is a different approach, where latent structures are sampled repeatedly during training and are 'rewarded' based on how closely their yield matches the gold graph.

A big next step is to generalize the AM algebra by applying it to semantic formalisms other than AMR. The AM algebra and our dependency model make some specific assumptions of how the meanings of words combine: that there is one connected graph fragment per word, that each such fragment has only one node (the root) where edges from other fragments can attach, and so on. It would be interesting to see which of these assumptions hold for other semantic representations, and where they do not, how the AM algebra could be adapted. A further, more practical challenge is that all heuristics developed in this thesis are specific for AMR and would need to be redeveloped for other representations. We saw how even a detail such as the source preference heuristic can have a sizable effect on parser performance, so it is important to get these right. This takes either serious manual effort, or the development of statistical methods as discussed above.

Despite these challenges, preliminary experiments that apply the AM algebra to semantic dependency parsing (using manual heuristics) showed promising results. Applications to semantic formalisms like Discourse Representation Theory (DRT, Kamp et al. (2011), Bos et al. (2017)), that are structurally more different from AMRs, show more substantial challenges. However, ongoing research by my colleagues Matthias Lindemann and Meaghan Fowlie indicates that these challenges may well be overcome (personal discussion, 2018).

7.2 A comeback of linguistic structure?

Around the time when work on this thesis started, the neural fever broke out. Neural networks led to simple, easy to implement models for all sorts of tasks in computational linguistics, and they were unreasonably good at it. It was a revolution, the whole field shifted, and end-to-end neural implementations popped up for nearly every task.

At this year's ACL, I felt like there might be a bit of a counterrevolution going on. Not that neural networks are getting pushed out; they are way too useful and effective for that. But to examine more closely what they do and do not encode (e.g. Conneau et al. (2018)), how they use linguistic structure (e.g. Liu et al. (2018)), or incorporating linguistic structure into neural models (e.g. Kuncoro et al. (2018), or, if I may, Groschwitz et al. (2018)). At the Workshop on Relevance of Linguistic Structure in Neural Architectures for NLP (RELNLP), Chris Dyer talked about how neural networks have the wrong inductive bias, and how to change that. Generally, at that workshop, the spirit seemed to be that linguistic structure is, and will become more, relevant. I thought this might just be due to the crowd for this particular workshop, but the next day, at the 3rd Workshop on Representation Learning for NLP (RepL4NLP), the spirit was similar.

So, linguistic structure might stage a comeback. And, what's more, everyone seems excited about it. I sure am excited. But why is that? Shouldn't I rather be excited that neural networks provide simple and effective solutions? To some extent it may just be rooting for the underdog. Or a deeply held conviction that the usefulness of linguistic structure is a truth, and now that truth breaks through. But I think that, at least for me, it's something different.

As computational linguists, we are engineers, scientists, and humans, all at once. As an engineer I pursue performance, my job is to build the model with the best evaluation results. As a scientist I pursue the truth, my job is to find out what works and why, or why not. But as a human, I want to do what I love. And what I love about computational linguistics is the interplay of language, mathematics and engineering. To input a sentence, watch the gears turn, and see something meaningful come out. And watching the gears turn is much more interesting when you actually understand what's going on. With neural networks, it is hard for me to see beyond the engineering part, to see the language and mathematics inside of them.

Thus, the thought that there may be more linguistics in the future of CL, more explicit structure, that thought is a promise that the engineer, the scientist, and the human in me will work towards the same goals. And that's exciting.

Bibliography

Artzi, Y., Lee, K., and Zettlemoyer, L. (2015). Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., and Schneider, N. (2013). Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186.

Bos, J., Basile, V., Evang, K., Venhuizen, N. J., and Bjerva, J. (2017). The Groningen meaning bank. In *Handbook of Linguistic Annotation*, pages 463–496. Springer.

Cai, S. and Knight, K. (2013). Smatch: An evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 748–752.

Chiang, D., Andreas, J., Bauer, D., Hermann, K. M., Jones, B., and Knight, K. (2013). Parsing graphs with Hyperedge Replacement Grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 924–932.

Chomsky, N. (1981). *Lectures on Government and Binding.* Foris.

Chu, Y.-J. (1965). On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400.

Clark, S. and Curran, J. R. (2004). The importance of supertagging for wide-coverage CCG parsing. In *Proceedings of the 20th International Conference on Computational Linguistics*, page 282. Association for Computational Linguistics.

Cohn, T., Goldwater, S., and Blunsom, P. (2009). Inducing compact but accurate tree-substitution grammars. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 548–556. Association for Computational Linguistics.

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the Eighth Conference on European Chapter of the Association for Computational Linguistics*, pages 16–23. Association for Computational Linguistics.

Comon, H. (1997). *Tree automata techniques and applications.*

Conneau, A., Kruszewski, G., Lample, G., Barrault, L., and Baroni, M. (2018). What you can cram into a single \$&!# vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136. Association for Computational Linguistics.

Copestake, A., Flickinger, D., Pollard, C., and Sag, I. A. (2005). Minimal recursion semantics: An introduction. *Research on Language and Computation*, 3(2-3):281–332.

Copestake, A., Lascarides, A., and Flickinger, D. (2001). An algebra for semantic construction in constraint-based grammars. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 140–147. Association for Computational Linguistics.

Courcelle, B. and Engelfriet, J. (2012). *Graph structure and monadic second-order logic: A language-theoretic approach*, volume 138. Cambridge University Press.

Dalrymple, M., Lamping, J., Pereira, F., and Saraswat, V. (1997). Quantifiers, anaphora, and intensionality. *Journal of Logic, Language and Information*, 6(3):219–273.

Damonte, M., Cohen, S. B., and Satta, G. (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*, volume 1, pages 536–546.

Diestel, R. (2018). *Graph theory.* Springer Publishing Company, Incorporated.

Dohare, S., Gupta, V., and Karnick, H. (2018). Unsupervised semantic abstractive summarization. In *Proceedings of ACL 2018, Student Research Workshop*, pages 74–83. Association for Computational Linguistics.

Dong, L. and Lapata, M. (2016). Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 33–43.

Dozat, T. and Manning, C. D. (2017). Deep biaffine attention for neural dependency parsing. *ICLR2017*.

Drewes, F., Kreowski, H.-J., and Habel, A. (1997). Hyperedge replacement graph grammars. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 1: Foundations*, pages 95–162. World Scientific.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards, B*, 71:233–240.

Eisner, J. (1996). Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 79–86. Association for Computational Linguistics.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464. Association for Computational Linguistics.

Flanigan, J., Dyer, C., Smith, N. A., and Carbonell, J. (2016). CMU at SemEval-2016 task 8: Graph-based AMR parsing with infinite ramp loss. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1202–1206.

Flanigan, J., Thomson, S., Carbonell, J., Dyer, C., and Smith, N. A. (2014). A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1426–1436.

Floyd, R. W. (1962). Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345.

Foland, W. and Martin, J. H. (2017). Abstract Meaning Representation parsing using LSTM recurrent neural networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 463–472.

Gamut, L. (1991). *Logic, language, and meaning, Volume 2: Intensional logic and intensional grammar*. Chicago: Univ. of Chicago Press.[LTF Gamut is the collective pseudonym for JFAK van Benthem, JAG Groenendijk, DHJ de Jong, MJB Stockhof, and HJ Verkuyl.].

Gontrum, J., Groschwitz, J., Koller, A., and Teichmann, C. (2017). Alto: Rapid prototyping for parsing and translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 29–32.

Groschwitz, J., Koller, A., and Johnson, M. (2016). Efficient techniques for parsing with tree automata. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2042–2051. Association for Computational Linguistics.

Groschwitz, J., Koller, A., and Teichmann, C. (2015). Graph parsing with s-graph grammars. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1481–1490. Association for Computational Linguistics.

Groschwitz, J., Lindemann, M., Fowlie, M., Johnson, M., and Koller, A. (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)*, pages 1831–1841. Association for Computational Linguistics.

Hall, D., Durrett, G., and Klein, D. (2014). Less grammar, more features. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 228–237.

Henning, S. (2017). Effizientes parsing von TAG mit supertagging. *Bachelor Thesis at Saarland University.*

Hockenmaier, J. and Bisk, Y. (2010). Normal-form parsing for Combinatory Categorial Grammars with generalized composition and type-raising. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 465–473.

Hopcroft, J. E. and Tarjan, R. E. (1973). Efficient Algorithms for Graph Manipulation [h] (algorithm 447). *Commun. ACM*, 16(6):372–378.

Hovy, E., Marcus, M., Palmer, M., Ramshaw, L., and Weischedel, R. (2006). OntoNotes: the 90% solution. In *Proceedings of the human language technology conference of the NAACL, Companion Volume: Short Papers*, pages 57–60. Association for Computational Linguistics.

Hoyt, F. and Baldridge, J. (2008). A logical basis for the D combinator and normal form in CCG. *Proceedings of ACL-08: HLT*, pages 326–334.

Issa, F., Damonte, M., Cohen, S. B., Yan, X., and Chang, Y. (2018). Abstract Meaning Representation for paraphrase detection. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, volume 1, pages 442–452.

Johnson, M. (1987). Grammatical relations in attribute-value grammars. In *Proceedings, West Coast Conference on Formal Linguistics*, volume 6, pages 103–114.

Jones, B., Andreas, J., Bauer, D., Hermann, K. M., and Knight, K. (2012). Semantics-based machine translation with Hyperedge Replacement Grammars. *Proceedings of COLING 2012*, pages 1359–1376.

Joshi, A. K. and Schabes, Y. (1997). Tree-adjoining grammars. In *Handbook of formal languages*, pages 69–123. Springer.

Kamp, H., Van Genabith, J., and Reyle, U. (2011). Discourse representation theory. In *Handbook of Philosophical Logic*, pages 125–394. Springer.

Kaplan, R. M., Bresnan, J., et al. (1982). Lexical-functional grammar: A formal system for grammatical representation. *Formal Issues in Lexical-Functional Grammar*, (47):29–130.

Kingsbury, P. and Palmer, M. (2002). From TreeBank to PropBank. In *LREC*, pages 1989–1993. Citeseer.

Kiperwasser, E. and Goldberg, Y. (2016). Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

Koller, A. (2015). Semantic construction with graph grammars. In *Proceedings of the 11th International Conference on Computational Semantics*, pages 228–238.

Koller, A. and Kuhlmann, M. (2011). A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 2–13. Association for Computational Linguistics.

Koller, A. and Kuhlmann, M. (2012). Decomposing TAG algorithms using simple algebraizations. In *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+ 11)*, pages 135–143.

Kracht, M. (2011). *Interpreted languages and compositionality*, volume 89. Springer Science & Business Media.

Kuncoro, A., Dyer, C., Hale, J., Yogatama, D., Clark, S., and Blunsom, P. (2018). LSTMs can learn syntax-sensitive dependencies well, but modeling structure makes them better. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1426–1436. Association for Computational Linguistics.

Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., and Steedman, M. (2010). Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 conference on empirical methods in natural language processing*, pages 1223–1233. Association for Computational Linguistics.

Lewis, M., Lee, K., and Zettlemoyer, L. (2016). LSTM CCG parsing. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 221–231.

Lewis, M. and Steedman, M. (2014). A* CCG parsing with a supertag-factored model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 990–1000. Association for Computational Linguistics.

Lindemann, M. (2018). AMR parsing as typed dependency parsing with a graph algebra. *Bachelor Thesis at Saarland University.*

Liu, N. F., Levy, O., Schwartz, R., Tan, C., and Smith, N. A. (2018). Lstms exploit linguistic attributes of data. In *Proceedings of The Third Workshop on Representation Learning for NLP*, pages 180–186.

Lyu, C. and Titov, I. (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)*, pages 397–407. Association for Computational Linguistics.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60.

Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41.

Montague, R. (1973). The proper treatment of quantification in ordinary English. In *Approaches to natural language*, pages 221–242. Springer.

Peng, X. and Gildea, D. (2016). UofR at SemEval-2016 task 8: Learning synchronous hyperedge replacement grammar for AMR parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1185–1189.

Peng, X., Song, L., and Gildea, D. (2015). A synchronous Hyperedge Replacement Grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*, pages 32–41.

Peng, X., Wang, C., Gildea, D., and Xue, N. (2017). Addressing the data sparsity issue in neural AMR parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 366–375. Association for Computational Linguistics.

Pennington, J., Socher, R., and Manning, C. (2014). GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Pollard, C. and Sag, I. A. (1994). *Head-driven phrase structure grammar*. University of Chicago Press.

Pourdamghani, N., Gao, Y., Hermjakob, U., and Knight, K. (2014). Aligning English strings with Abstract Meaning Representation graphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 425–429.

Rao, S., Marcu, D., Knight, K., and Daumé III, H. (2017). Biomedical event extraction using Abstract Meaning Representation. *BioNLP 2017*, pages 126–135.

Shieber, S. M., Schabes, Y., and Pereira, F. C. (1995). Principles and implementation of deductive parsing. *The Journal of logic programming*, 24(1-2):3–36.

Steedman, M. (2000). *The syntactic process*. MIT press.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Szabó, Z. G. (2017). Compositionality. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2017 edition.

Teichmann, C., Venant, A., and Koller, A. (2018). Efficient translation with linear bimorphisms. In *International Conference on Language and Automata Theory and Applications*, pages 308–320. Springer.

van Noord, R. and Bos, J. (2017). Neural semantic parsing by character-based translation: Experiments with Abstract Meaning Representations. *Computational Linguistics in the Netherlands Journal*, 7:93–108.

Vinyals, O., Kaiser, Ł., Koo, T., Petrov, S., Sutskever, I., and Hinton, G. (2015). Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781.

Wang, C., Pradhan, S., Pan, X., Ji, H., and Xue, N. (2016). CAMR at SemEval-2016 task 8: An extended transition-based AMR parser. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1173–1178.

Wang, C., Xue, N., and Pradhan, S. (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*, pages 366–375.

Wang, Y., Liu, S., Rastegar-Mojarad, M., Wang, L., Shen, F., Liu, F., and Liu, H. (2017). Dependency and AMR embeddings for drug-drug interaction extraction from biomedical literature. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 36–43. ACM.

A

Proofs

A.1 Proofs for Chapter 4

Lemma 4.4: *Let H a sub-s-graph of G that is a reachable state in D. Then H is an extensible state in D if and only if its edge set is the union of a set of s-components of G and all source nodes in G that are nodes of H are also source nodes in H, i.e. $\mathit{Src}\,(G) \cap V_H \subseteq \mathit{Src}\,(H)$.*

Proof of Lemma 4.4. First, let us assume that the edge set E_H of H is indeed the union of a set of s-components (with respect to ϕ_H), and $\mathit{Src}\,(H) \subseteq \mathit{Src}\,(G)$. In fact, as we observed earlier, the graph H° is induced by E_H, since H is reachable. Let now $K = (K^\circ, \phi_K)$ be the s-graph where K° is the graph induced by $E_G \setminus E_H$ and $\phi_K = \phi_H|_{V_K} \cup \phi_G|_{V_G \setminus V_H}$. That is, we use all the edges not in H, use the sources of H where the graphs overlap and the sources of G elsewhere. In fact, since E_H is the union of a set of s-components, the same holds for K, and it can be easily seen that the graphs can then only overlap in source nodes, and there the sources agree. Thus, $H \mid\mid K$ is defined, and the result covers all edges (and thus vertices) of G. Further, we ensured that all source nodes of G are also source nodes of $H \mid\mid K$. Then let fg_B be the operation that forgets all sources on nodes that are sources nodes in $H \mid\mid K$ but not in G, and let ren_h be the operation that renames the sources in $\mathit{fg}_B\,(H \mid\mid K)$ to what they are in G. Then $G = \mathit{ren}_h\,(\mathit{fg}_B\,(H \mid\mid K))$, i.e. H is extensible.

Let now H be extensible, i.e. there is a term t with $t \xrightarrow[D]{*} G$ that has H as a leaf. we need to show that H is the union of a set of s-components of G, and that all source nodes in G that are nodes of H are also source nodes in H.

Let us first assume for a contradiction that H is not the union of a set of s-components. That is, one of the s-components must be in H *partially*. I.e. there are two edges e, f with $e \sim_{D(\phi_H)} f$ such that $e \in E_H, f \notin E_H$. Now, $e \sim_{D(\phi_H)} f$

means that there is a path $v_1 \overset{e}{\leftrightarrow} v_2 \leftrightarrow \ldots v_{k-1} \overset{f}{\leftrightarrow} v_k$ with none of $v_2, \ldots, v_{k-1}$ in $D(\phi_H)$. Since $e \in E_H$ but $f \notin E_H$, along this path there must be neighboring edges e', f' such that still $e' \in E_H, f' \notin E_H$ (we just go along the edges of the path until we hit the first edge f' not in E_H). Remember that the vertex v_i between e' and f' is not a source node in H. Now we consider a fixed run r on t that yields G, and move upwards through the term t, starting at the leaf H, until the edge f' is added by a merge operation. More precisely, there must be a merge operation, such that one of its children K contains e' and the other child L contains f'. Since H did not have a source at v_i, neither can K (none of the operations *add* source names to a node), but clearly, v_i is in both K and L. This is a contradiction to Definition 2.11, merging K and L is not allowed, and thus such a run r cannot exist, a contradiction to H being extensible.

Now let us assume that some source nodes in G that are nodes of H are not source nodes in H. This yields a contradiction even faster, since no operation can add source names to a node and we can therefore never obtain G from H with operations of the HR algebra. □

Lemma 4.5: *Let $\mathcal{C} = (C, \phi_{\mathcal{C}})$, $\mathcal{C}_1 = (C_1, \phi_{\mathcal{C}_1})$, $\mathcal{C}_2 = (C_2, \phi_{\mathcal{C}_2})$ be s-component representations in the s-graph G. Then $T(\mathcal{C}) = T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ if and only if*

(i) $\phi_{\mathcal{C}} = \phi_{\mathcal{C}_1} \cup \phi_{\mathcal{C}_2}$ (in particular, $\phi_{\mathcal{C}_1}$ and $\phi_{\mathcal{C}_2}$ must be compatible).

(ii) All sets in C_1 and C_2 are s-components with respect to $D(\phi_{\mathcal{C}})$.

(iii) $C = C_1 \uplus C_2$ (i.e., disjoint union).

If there is no such $\mathcal{C}$, then $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ is not defined.

Proof of Lemma 4.5. For brevity, let us write $H = T(\mathcal{C})$, $K = T(\mathcal{C}_1)$ and $L = T(\mathcal{C}_2)$. Recall the conditions of Definition 2.11 for the merge of concrete s-graphs, with this choice of graphs:

1. $K \subseteq H$, $L \subseteq H$, $H^\circ = K^\circ \cup L^\circ$,
2. $V_K \cap V_L = \mathit{Src}(K) \cap \mathit{Src}(L)$
3. $E_K \cap E_L = \emptyset$
4. $\phi_H = \phi_K \cup \phi_L$.

We need to show that these conditions hold if and only if (i) to (iii) from the lemma hold. Let us first assume that (i) to (iii) hold. Then (4) follows immediately. Conditions (ii) and (iii) give us that K° and L° are induced by disjoint edge sets, and H° is induced by the union of those sets. This gives us (1) and (3). To see that (2) holds, note that the edges of C_1 and C_2 meet where different s-components (with respect to both $D(\phi_{\mathcal{C}_1})$ and $D(\phi_{\mathcal{C}_2})$) meet. There, by the definition of s-components, we must have sources, and (2) follows.

Now assume that (1)-(4) hold. Again, (i) follows immediately, and if we had (ii), then (iii) would follow from (1) – that $C = C_1 \cup C_2$ – and (3) –that the union is disjoint. It remains to show (ii). First note that all source nodes defined by $\phi_{\mathcal{C}_1}$ are also source names in $\phi_{\mathcal{C}}$. Therefore, the former partitions the edges more finely into s-components than the latter, i.e. the only way an s-component $M \in C_1$ is not also an s-component with respect to $D(\phi_{\mathcal{C}})$ is if it is a disjoint union of s-components $N_1, \ldots, N_k$ with respect to $D(\phi_{\mathcal{C}})$. Something must split these N_i apart, that is, there must be a source node v in $D(\phi_{\mathcal{C}})$ that is not a source in $\mathcal{C}_1$, i.e. $v \notin Src(K)$. The node v must thus be a source in $\mathcal{C}_2$, i.e. a vertex in V_L. But since its surrounding edges are in M –that is how we found v in the first place, v must be in V_K as well. But $v \notin Src(K)$, a contradiction to Condition (2). Thus, (ii) must hold. The same argument can be made with the roles of $\mathcal{C}_1$ and $\mathcal{C}_2$ reversed.

The above arguments also show that if $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ is defined, i.e. if (1)-(4) hold, then $\mathcal{C}$is well-defined via (i) and (iii). □

Lemma 4.7: *Let G be an s-graph, let $\beta = (E, \phi), \beta_1 = (E_1, \phi_{\beta_1}), \beta_2 = (E_2, \phi_{\beta_2})$ be boundary representations in G. Then $T(\beta) = T(\beta_1) \mid\mid T(\beta_2)$ if and only if the following conditions hold:*

(i) $\phi = \phi_{\beta_1} \cup \phi_{\beta_2}$*;*

(ii) for every source node v of β_1 that is not a source node in β_2, the last edge on the shortest path in G from v to the closest source node of β_2 is not an in-boundary edge of β_2, and vice versa;

(iii) $E = E_1 \uplus E_2$*, i.e. the disjoint union.*

We write $\beta = \beta_1 \mid\mid \beta_2$. If no such β exists, $T(\beta_1) \mid\mid T(\beta_2)$ is undefined.

Proof of Lemma 4.7. For brevity, let us write $H = T(\beta)$, $K = T(\beta)$ and $L = T(\beta)$. We need to show that $H = K \mid\mid L$ if and only if Conditions (i) to (iii) hold.

Let us first assume that the conditions hold. In fact, we will then show that (i) to (iii) of Lemma 4.5 hold for the corresponding s-component representations. As in the proof of Lemma 4.5, the only non-trivial thing to prove is the criterion (ii). As in that proof, if we assume the existence of an s-component M specified by β_1 that is not an s-component with respect to $D(\phi_\beta)$, then we obtain a node v that is a source node in β_2, and a node but not a source node in β_1. And further, the edges adjacent to v are in M, and there is at least one such edge e. By Condition (ii) of this lemma, then the last edge f on the shortest path from v to the closest source node in β_1 is not in β_1. But by construction of f, we have $e \sim_{D\left(\phi_{\beta_1}\right)} f$ and thus f is in M and therefore in β_1, a contradiction. Thus, all conditions of Lemma 4.5 hold and $H = K \mid\mid L$ follows.

If we in turn assume that $H = K \mid\mid L$ holds, then we get Conditions (i) and (iii) of this lemma immediately from Definition 2.11 of merging concrete graphs. So let us assume that (ii) does not hold, and there is a source node v of β_1 that is not a source node of β_2, such that the last edge f on the shortest path from v to the closest source node u of β_2 is an edge in β_2. Since u is the *closest* source node in β_2, this shortest path does not pass a source node of β_2, and therefore any edge e incident to v (such an edge must exist) is in the same s-component (with respect to $D(\phi_{\beta_2})$) as f, i.e. $e \in [f]_{\sim_{D\left(\phi_{\beta_2}\right)}}$. Thus e and therefore v must be in $L = T(\beta_2)$, but v is not a source node there by its definition. And yet, v was chosen to be in $K = T(\beta_1)$, a contradiction to Condition (2) of Definition 2.11.

The above arguments also show that if $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ is defined, i.e. if (1)-(4) hold, then $\mathcal{C}$is well-defined via (i) and (iii). □

A.2 Proofs for Chapter 6

Lemma 6.7: *Let $\prec$ be a partial order on a finite set $x_1, \dots, x_k$ and σ a permutation on $1, \dots, k$, such that both the sequences $x_1, \dots, x_k$ and $x_{\sigma(1)}, \dots, x_{\sigma(k)}$ respect the partial order $\prec$; i.e. if $x_i \prec x_j$, then $i < j$ and $\sigma(i) < \sigma(j)$. Then there is a sequence of adjacent transpositions $\mu_1, \dots, \mu_m$ such that $\sigma = \mu_m \circ \dots \circ \mu_1$ and for every $\ell = 1, \dots, M$, the intermediate sequence $x_{(\mu_\ell \circ \dots \circ \mu_1)(1)}, \dots, x_{(\mu_i \circ \dots \circ \mu_1)(k)}$ respects $\prec$; i.e. if $x_i \prec x_j$ then $(\mu_\ell \circ \dots \circ \mu_1)(i) < (\mu_\ell \circ \dots \circ \mu_1)(j)$.*

Proof. Let M be the number of pairs $i < j$ such that $\sigma(i) > \sigma(j)$, i.e. the number of pairs that are out of order in σ. We show the above statement by induction on M. If $M = 0$, then σ is the identity and the statement is trivially true.
Let now $M > 0$. Let j be such that there is a $i < j$ with $\sigma(i) > \sigma(j)$ and such that $\sigma(j)$ is maximal. Then let ℓ such that $\sigma(\ell) = \sigma(j) + 1$.
Claim: $\ell < j$.
Proof of claim: If $\ell > j$ then $\ell > j > i$. At the same time, $\sigma(\ell) > \sigma(j) > \sigma(i)$, which is a contradiction to j being chosen such that $\sigma(j)$ is maximal.
Let now π be the permutation that swaps $\sigma(j)$ and $\sigma(\ell) = \sigma(j) + 1$; i.e. π is an adjacent transposition. Furthermore, $\pi \circ \sigma$ is a permutation with only $M - 1$ pairs

out of order: only $\sigma(j)$ and $\sigma(\ell)$ have been swapped, and as we just saw, they were put in the right order.

Claim: The sequence $x_{(\pi\circ\sigma)(1)}, \ldots, x_{(\pi\circ\sigma)(k)}$ respects $\prec$.

Proof of claim: Since σ respects $\prec$ by definition, and π only swaps $\sigma(\ell)$ with $\sigma(j)$, we only need to consider the pair $(\pi\circ\sigma)(\ell), (\pi\circ\sigma)(j)$. In fact, we know that $(\pi\circ\sigma)(\ell) < (\pi\circ\sigma)(j)$, thus we only need to show that not $x_j \prec x_\ell$. But if $x_j \prec x_\ell$, then $j < \ell$ follows from the lemma's assumptions, which is a contradiction to $\ell < j$ that we showed before. In short, since π brings us closer to the original order, and the original order respects $\prec$, adding π does not interfere with $\prec$.

Thus, by induction, there is a sequence of adjacent transpositions $\mu_1, \ldots, \mu_m$ such that $\pi\circ\sigma = \mu_m \circ \ldots \circ \mu_1$ and for every $\ell = 1, \ldots, \mathsf{M}$, the intermediate sequence $x_{(\mu_\ell\circ\ldots\circ\mu_1)(1)}, \ldots, x_{(\mu_i\circ\ldots\circ\mu_1)(k)}$ respects $\prec$. Thus, we can write σ as the sequence of adjacent transpositions

$$\sigma = \pi^{-1} \circ \mu_m \circ \ldots \circ \mu_1$$

where as we just saw, also the intermediate sequence $\mu_m \circ \ldots \circ \mu_1 = \pi\circ\sigma$ respects $\prec$. $\square$

B

Aligner

The core principles of the aligner here are similar to the JAMR aligner of Flanigan et al. (2014). There are two types of actions:

Action 1: Align a node to a word, such as *bake-01* to *baker* and *whistle-01* to *whistles* in Figure B.1. This action is based on the word and the node label, using lexical similarity, handwritten rules[1] and WordNet neighborhood; we align some name and date patterns directly. The node is fixed as the lexical node of the alignment.

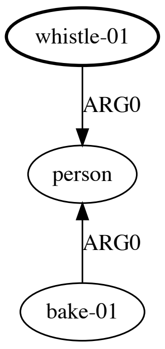


Figure B.1: AMR for *The baker whistles.*

Action 2: Extend an existing alignment to another node, such as from *bake-01* to *person* in Figure B.1. Such an extension is chosen on a heuristic based on

1. the direction and label of the edge along which the

[1]E.g. the node label *have-condition-91* can be aligned to *if* and *otherwise*.

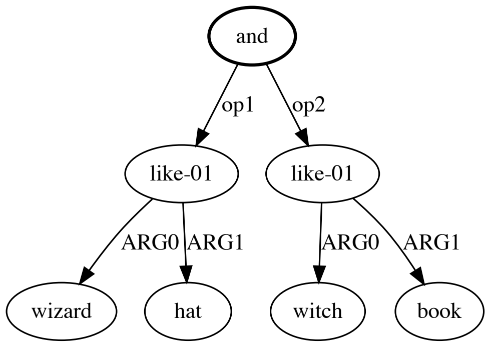


Figure B.2: AMR for *The wizard likes hats and the witch books.*

alignment is split,

2. the labels of both the node we spread from, and the node we spread to, and
3. the word in the alignment.

Note that in the AM algebra, edges from other constants can only connect to a given constant at one node, the root. We therefore disallow this extension process if it would lead to two nodes in the constant that have incident edges from outside the constant (i.e. incident edges not belonging to the blob of any node in the constant, c.f. Section 5.5). An exception to that is when one word generates two occurrences of the same graph constant in the AMR. An example is ellipsis, as in Figure B.2, where the word *likes* corresponds to both of the *like-01* nodes. We cannot handle these graphs with such 'duplicate' nodes properly with our model, and exclude them from the training set.

Each action has a basic heuristic score, which we increase if a nearby node is already aligned to a nearby word, and decrease if other potential operations conflict with this one. This gives us a confidence score for every possible action, taking into account that every node and word can participate in at most one alignment. We iteratively execute the action with highest confidence until all heuristic options are exhausted or all nodes aligned. We then align remaining unaligned nodes to words near adjacent alignments.

C

Postprocessing

Having obtained an AM dependency tree, we can recover an AM term and evaluate it. During postprocessing we have to re-lexicalize the resulting graph according to the input string. For relatively frequent words in the training data (occurring at least 10 times), we take the supertagger's prediction for the label. For rarer words, the neural label prediction accuracy drops, and we simply take the node label observed most often with the word in the training data. For unseen words, if the lexicalized node has outgoing ARGx edges, we first try to find a verb lemma for the word in Princeton WordNet Miller (1995) (we use version 3.0). If that fails, we try, again in WordNet, to find the closest verb derivationally related to any lemma of the word. If that also fails, we take the word literally. In any case, we add *-01* to the label. If the lexicalized node does not have an outgoing ARGx edge, we try fo find a noun lemma for the word in Princeton WordNet, and otherwise take the word literally.

D

NP-completeness of the typed decoding problem

Credit for this proof goes to Alexander Koller. The proof has been published before in Groschwitz et al. (2018) and is reprinted here only for convenience.

We prove NP-completeness for the well-typed decoding problem by reduction from HAMILTONIAN-PATH.

Let $G = (V, E)$ be a directed graph with nodes $V = \{1, \ldots, n\}$ and edges $E \subseteq V \times V$. A Hamiltonian path in G is a sequence $(v_1, \ldots, v_n)$ that contains each node of V exactly once, such that $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq n-1$. We assume w.l.o.g. that $v_n = n$. Deciding whether G has a Hamiltonian path is NP-complete.

Given G, we construct an instance of the decoding problem for the sentence $w = 1 \ldots n$ as follows. We assume that the first graph fragments shown in Fig. D.1a (with node label "i") is the only graph fragment the supertagger allows for 1, ..., $n-1$, and the second one (with node label "f") is the only graph fragment allowed for n. We let $\omega^{\mathsf{ex}}_{[i \to k]} = 1$ if $(i, k) \in E$, and zero otherwise.

Under this construction, every well-typed AM dependency tree for w corresponds to a linear sequence of nodes connected by edges with label $\textsc{app}_s$ (see Fig. D.1c for an example) More specifically, n is a leaf, and every node except for n has precisely one outgoing $\textsc{app}_s$ edge; this is enforced by the well-typedness. Because of the edge scores, the score of such a dependency tree is $n-1$ iff it only uses edges that also exist in G; otherwise the score is less than $n-1$. Therefore, we can decide whether G has a Hamiltonian path by running the decoder, i.e. computing the highest-scoring well-typed AM dependency tree t for w, and checking whether the score of t is $n-1$.

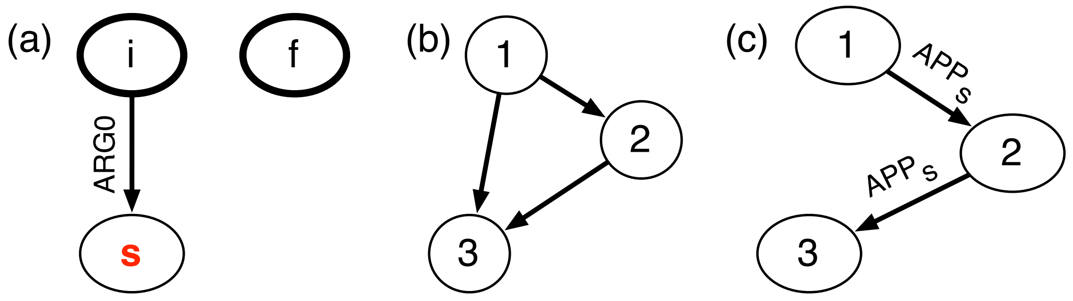


Figure D.1: (a) The two graph fragments required for the NP-completeness proof. (b) An example graph and (c) the AM dependency tree corresponding to its Hamiltonian path.

E

Neural network details.

1. As pre-trained embeddings, we use GloVE Pennington et al. (2014). The vectors we use have 200 dimensions and are trained on Wikipedia and Gigaword. We add randomly initialized vectors for the `name`, `date` and `number` tokens and for the unknown word token (if no GloVE vector exists). We keep these embeddings fixed and do not train them.

2. For the learned word embeddings, we follow K&G in all our models in using a word dropout of $\alpha = 0.25$. That is, during training, for a word that occurs k times in the training data, with probability $\frac{\alpha}{k+\alpha}$ we instead use the word embedding for the unknown word token instead of w_i.

3. The character-based encodings c_i for the supertagger are generated by a single layer LSTM with 100 hidden dimensions, reading the word left to right. If a word (or sequence of words) is replaced by e.g. a `name` token during preprocessing, the character-based encoding reads the original string instead (this helps to classify names correctly as country, person etc.).

4. To prevent overfitting, we add dropout of 0.5 in the LSTM layers of all the models except for the K&G model which we keep as implemented by the authors. We also add 0.5 dropout to the MLPs in the supertagger and local dependency model.

5. Further hyperparameters are detailed in Table E.1.

Optimizer	Adam
Learning Rate	0.004
Epochs	up to 100
Pre-trained word embeddings	glove.6B
Pre-trained word emb. dimension	200
Learned word emb. dimension	100
POS embedding dimension	32
Character encoding dimension	100
α (word dropout)	0.25
Bi-LSTM layers	2 (stacked)
Hidden dimensions in each LSTM	256
Hidden units in MLPs	256
Internal dropout of LSTMs, MLPs	0.5
Input vector dropout	0.8

Table E.1: Hyperparameters used for the neural models in Chapter 6.