Universität des Saarlandes
Fachrichtung Sprachwissenschaft und Sprachtechnologie

MSc Thesis in Language Science and Technology

Fast transition-based AM dependency parsing with well-typedness guarantees

Autor:
Matthias Lindemann
Matrikelnummer: 2558608

Gutachter:
Prof. Dr. Alexander Koller
Dr. Jonas Groschwitz

August 4, 2020

Eidesstattliche Erklärung
Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Ich versichere, dass die gedruckte und die elektronische Version der Masterarbeit inhaltlich übereinstimmen.

Saarbrücken, den 4. August 2020.

Matthias Lindemann

Acknowledgements

I thank Alexander Koller for his advice and for giving me so much freedom in pursing this research. His initial skepticism towards the idea of this thesis but continuous support helped me grow as a researcher.
I thank Jonas Groschwitz for his advice, comments on drafts and so many fun, interesting and stimulating discussions during this project and more generally in the last two years.
Finally, I thank Christine for her comments on a draft of this thesis and reminding me of doing something else every once in a while.

Contents

Chapter 1

Introduction

Representing meaning formally and automatically deriving meaning representations from sentences have attracted considerable attention recently because this not only gives a computational perspective on semantics and its interface to syntax but also has practical applications of economic interest, such as conversational agents.

Often these formal meaning representations take the form of semantic graphs, that is, graphs that focus on representing predicate-argument-structure. Examples of such meaning representations are AMR (Banarescu et al., 2013), EDS (Oepen and Lønning, 2006), DM, PAS, PSD (Oepen et al., 2015) and UCCA (Abend and Rappoport, 2013).

Among the state-of-the-art approaches to semantic graph parsing is AM dependency tree parsing, introduced by Groschwitz et al. (2018) for AMR. Its particular strength lies in being applicable to many styles of semantic graphs. Since its introduction it has been used for parsing into DM, PAS, PSD, EDS (Lindemann et al., 2019) and UCCA (Donatelli et al., 2019). The basic idea of AM dependency parsing is to predict a term of the AM algebra (Groschwitz et al., 2017; Groschwitz, 2019), which then deterministically evaluates to a semantic graph. A practical challenge with AM dependency parsing is that the parsing problem is NP-complete (see Groschwitz et al. (2018)) due to the linguistically-inspired type constraints of the AM algebra. Groschwitz et al. present two parsing algorithms: one limits the search-space to projective dependency trees and has run time $O(n^5)$ in the length of the sentence, while the other also allows non-projective dependency trees but has worst-case exponential run time in the maximal out-degree of the tree.

For AM dependency parsing to be applicable in scenarios where computation time is a limited resource, such as parsing web-scale corpora, or treating the AM dependency tree as a latent variable in an end-to-end approach (Yogatama et al., 2017), these parsing algorithms are too slow. Theoretical work by Venant and Koller (2019) also shows that a formalism with bounded memory like the AM algebra is more expressive if non-projective dependency structures are allowed. This underlines the need for fast, polynomial time non-projective AM dependency parsers.

In this thesis, we develop transition-based parsing algorithms for non-projective AM dependency parsing inspired by the syntactic dependency parser of Ma et al. (2018) that are faster in practice than the existing AM dependency parsers and only take quadratic time in the length of the sentence. At the same time, they still guarantee well-typedness of the derived tree and that we can always finish a derivation without having to backtrack. We empirically verify that the transition-based parsers are not only faster but also on par with or even better than existing parsing algorithms in terms of accuracy.

Substantial parts of this thesis are currently under review as Lindemann et al. (2020).

Chapter 2

Background

2.1 Semantic graphs

Semantic graphs are a means of representing sentence meaning, and usually focus on capturing predicate-argument structure, that is, semantic graphs seek to capture the answers to who did what to whom for a given sentence.

Representing meaning accurately is a great challenge and there are many approaches to representing predicate-argument structures using graphs, drawing from different linguistic traditions and with different objectives. As a consequence, there are a number of different graphbanks, that is, corpora of sentences annotated with their semantic graphs, available for English, each following different design principles.

Figure 2.1 shows examples of semantic graphs in the representation styles of DM, PAS and PSD (Oepen et al., 2015). Figures 2.2a and 2.2b show semantic graphs for the same sentence from the perspective of EDS (Oepen and Lønning, 2006) and AMR (Banarescu et al., 2013), respectively. Note that some nodes have more than one incoming edge, which would not be allowed in a tree. We call an edge pointing to a node that already has another incoming edge a re-entrancy.

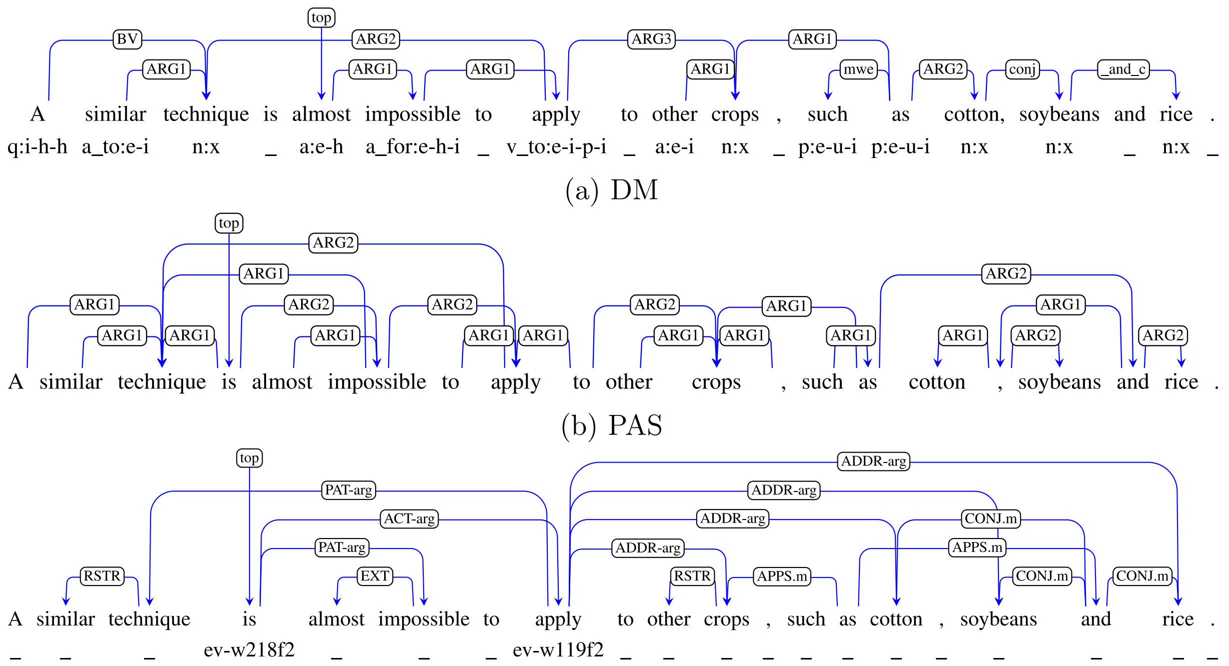
Formally speaking, a semantic graph is a labeled, directed graph where the edges represent the relations expressed in the sentence and nodes correspond to predicates or concepts – depending on the formalism. Usually, exactly one node is designated as top node, or root, identifying the main predicate (or highest scoping predicate) of the sentence.

The representations differ not only in their linguistic annotation but also in their formal properties. For example, DM, PAS and PSD graphs are acyclic, whereas EDS and AMR graphs can contain cycles for specific phenomena. Another major difference is the relationship of the semantic graph to its sentence. In DM, PAS and PSD, every node of the graph corresponds to a token whereas in EDS each node is *anchored* in a (character) span, e.g. the *udef_q node* with the BV edge to *_crop_n_1* is anchored in $\langle 53 : 100 \rangle$, which corresponds to "other crops, such as cotton, soybeans and rice.". Finally, AMR does not annotate any direct relationship between the nodes of the graph and the tokens in the sentence.

See Oepen et al. (2019) for a more in-depth discussion of the formal and linguistic differences between these semantic graphs.

2.2 AM algebra

The Apply-Modify algebra (Groschwitz et al., 2017; Groschwitz, 2019), also for short AM algebra, is an algebra for semantic graphs that follows linguistic principles. As algebra, it



top
BV
ARG1
ARG2
ARG3
mwe
conj
_and_c
A similar technique is almost impossible to apply to other crops , such as cotton, soybeans and rice .
q:i-h-h a_to:e-i n:x _ a:e-h a_for:e-h-i _ v_to:e-i-p-i _ a:e-i n:x _ p:e-u-i p:e-u-i n:x n:x _ n:x _
A similar technique is almost impossible to apply to other crops , such as cotton , soybeans and rice .
RSTR
PAT-arg
ACT-arg
EXT
ADDR-arg
CONJ.m
APPS.m
A similar technique is almost impossible to apply to other crops , such as cotton , soybeans and rice .
_ _ _ ev-w218f2 _ _ _ ev-w119f2 _ _ _ _ _ _ _ _ _ _ _ _

(a) DM

(b) PAS

(c) PSD

Figure 2.1: Examples of semantic graphs, adapted from Oepen et al. (2015). Annotations below the tokens provide further frame and word-sense information.

makes the compositional nature of sentence meaning explicit and builds semantic graphs along that compositional structure. The AM algebra has been successfully applied to several graphbanks, including DM, PAS, PSD, EDS and AMR (Lindemann et al., 2019) and UCCA (Donatelli et al., 2019).

In this section, we provide an introduction to the AM algebra that explains the main concepts but makes a few simplifications. See Groschwitz (2019) for the full details. The rest of this thesis and the implementation assumes the version of Groschwitz (2019).

Before we review the AM algebra, we clarify what we mean with algebra. We use a definition that is essentially that of Koller and Kuhlmann (2011):

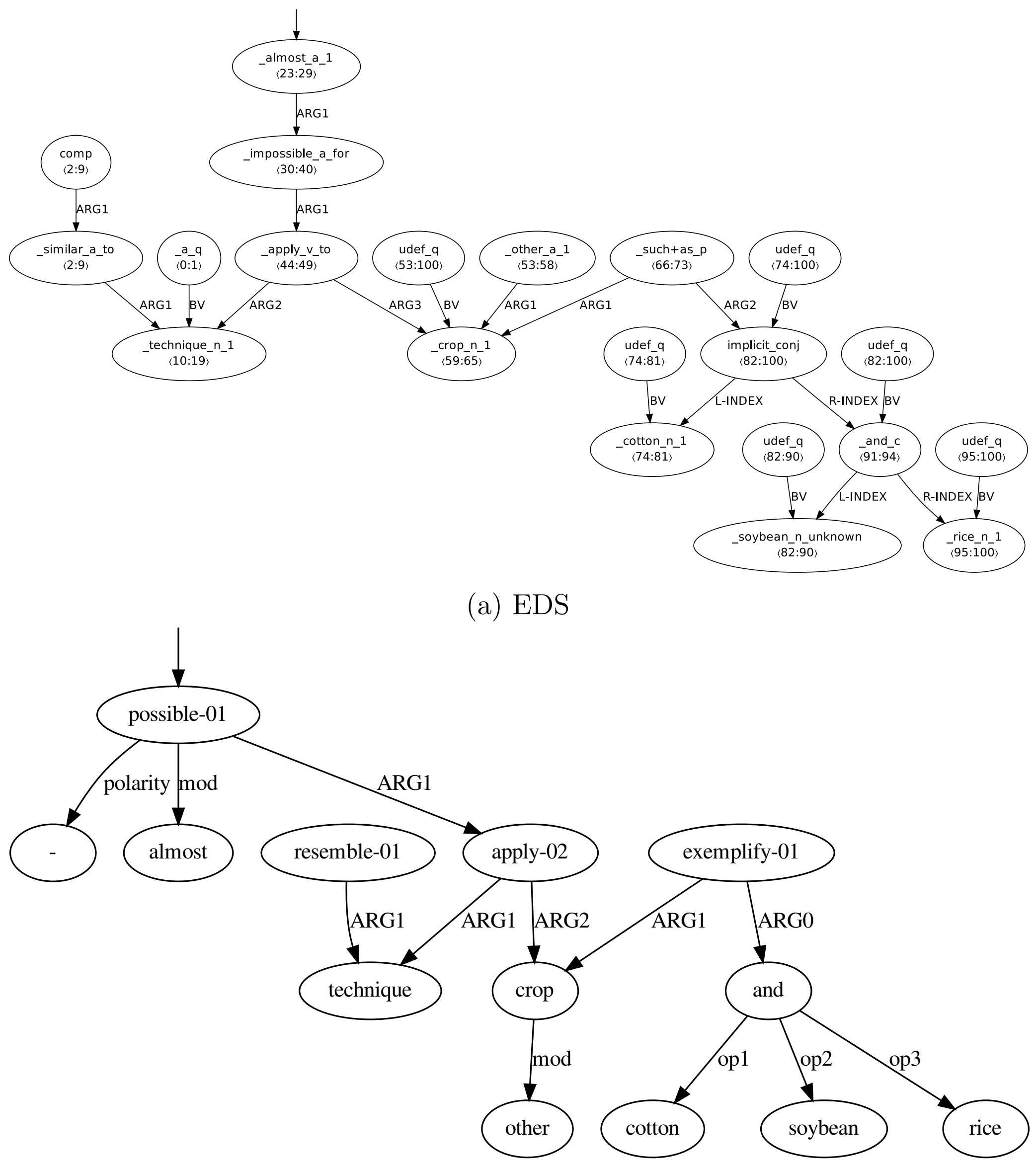
Algebra An algebra is a pair $\langle A, \Sigma \rangle$, where A is a set called the *domain* of the algebra (in our case: semantic graphs) and Σ is a set of function symbols called the *signature*, where each function symbol $\textsc{f} \in \Sigma$ is associated with a function f and a rank $r \in \mathbb{N}$ describing the arity of f. Function symbols with arity 0 are called constants and are interpreted as elements of A.

A *term* of an algebra with signature Σ is a tree over the symbols $\textsc{f} \in \Sigma$ such that the number of children of each node matches the rank of the function symbol. Terms can be *evaluated* recursively, denoted by $[\![\cdot]\!]$, by applying the functions to their arguments:

$$[\![\textsc{f}(t_1, t_2, \ldots, t_n)]\!] = [\![\textsc{f}]\!]([\![t_1]\!], [\![t_2]\!], \ldots, [\![t_n]\!]) = f([\![t_1]\!], [\![t_2]\!], \ldots, [\![t_n]\!])$$

Note that this notion of evaluation is a way of phrasing the principle of compositionality.

as-graphs The domain of the AM algebra is the set of *annotated graphs with sources* (as-graphs, for short), which build on s-graphs (Courcelle and Engelfriet, 2012). An s-graph is a graph where every node can be marked with a *source name* s from a fixed and finite set $\mathcal{S}$. The node that is marked with the source name s is called the s-source. In an s-graph, every source name $s \in \mathcal{S}$ can occur at most once. The as-graphs that the AM



(a) EDS

(b) AMR.

Figure 2.2: EDS and AMR graphs for "A similar technique is almost impossible to apply to other crops, such as cotton, soybeans and rice."; adapted from Oepen et al. (2015).

algebra uses are s-graphs where source names can have further annotations and one node is marked with a root-source name; the root-source plays a special role in how the AM algebra builds semantic graphs. Figure 2.3 shows examples of as-graphs in the style of AMR for the words *see*, *want*, *writer*, *sleep*, *soundly* and *sea*. Sources are marked in red, except for the root-source which can be identified by the bold outline.

From a linguistic perspective, a source can be seen as a placeholder or slot that must be filled with an argument or can act as modifier.

Operations of the AM algebra The signature of the AM algebra consists of graph constant symbols and binary operation symbols $\textsc{App}_\alpha$ and $\textsc{mod}_\beta$, where α and β are source names and further parameterize the operations.

The $\textsc{mod}_\beta(G, H)$ operation takes the β-source of the modifier H and merges it with the root-source of the head G; the β source name disappears. If G and H have other source names in common, the corresponding sources merge. Figure 2.4(b) shows an example of

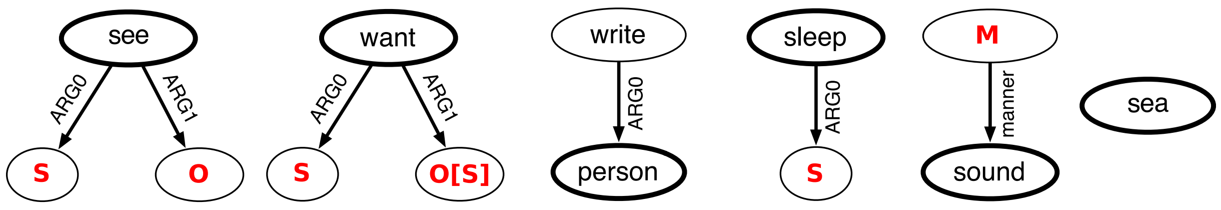


Figure 2.3: as-graphs $G_{\text{see}}, G_{\text{want}}, G_{\text{writer}}, G_{\text{sleep}} G_{\text{soundly}}, G_{\text{sea}}$.

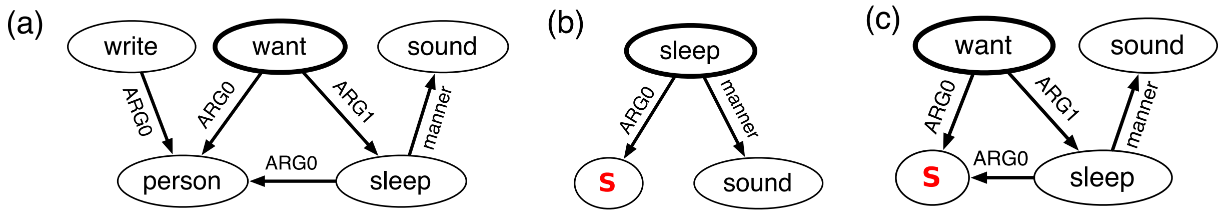


Figure 2.4: (a) AMR graph for "the writer wants to sleep soundly", (b) $[\![\text{MOD}_m(G_{\text{sleep}}, G_{\text{soundly}})]\!]$ and (c) $[\![\text{APP}_o(G_{\text{want}}, \text{MOD}_m(G_{\text{sleep}}, G_{\text{soundly}}))]\!]$.

$\text{MOD}_M(G_{\text{sleep}}, G_{\text{soundly}})$. Note that this operation can be repeated, capturing the property that a head can be modified as many times as desired.

The other operation is $\text{APP}_\alpha(G, H)$, which fills the α-source of the head G with the root-source of the argument H. That is, in the resulting graph, the root-source of H becomes the same node as the α-source of m and this node loses both source names. If G and H have other source names in common, the corresponding sources merge as well. Figure 2.4 (c) shows the result of of $\text{APP}_o(G_{\text{want}}, G_{\text{sleep-soundly}})$ where $G_{\text{sleep-soundly}} = [\![\text{MOD}_M(G_{\text{sleep}}, G_{\text{soundly}})]\!]$ is the partial result obtained before.

In the resulting graph of an APP_α operation there is no α-source anymore, so the operation cannot be repeated; the slot is filled, which models the property of heads to select exactly one argument of a certain type (e.g. a finite verb selects exactly one subject).

We can build the entire AMR graph in Figure 2.4a by evaluating a term of the AM algebra (AM term, for short) that makes use of constant symbols that denote the as-graphs in Figure 2.4. The AM term is shown in Figure 2.5a.

Types Source names can be annotated with *requests*, which state a request about what the *type* of the argument should be. The type of an as-graph is the set of its source names and their annotations[1]. For example the o-source in Figure 2.3 is annotated with $[s]$, so the type $\tau(G_{\text{want}})$ of G_{want} is $[s, o[s]]$ while the type of $\tau(G_{\text{see}})$ is $[s, o]$. We write $req_\alpha(\tau(G))$ for the request at α of graph G. The type of a graph without sources is the empty type, $[\,]$. If a source name is not explicitly annotated, the request is understood to be the empty type, so $[s, o]$ is understood as $[s[\,], o[\,]]$.

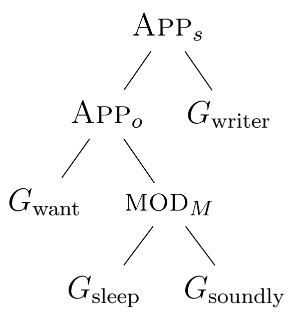
The types restrict which operations are allowed. In particular, the requests must be always met. More formally, $\text{APP}_\alpha(G, H)$ is allowed if and only if

(i) G has an α-source

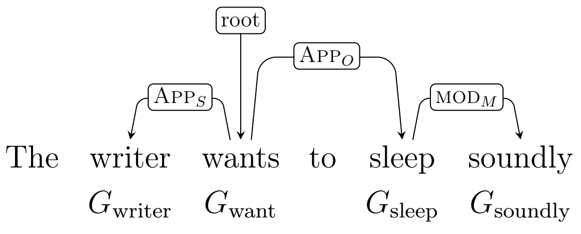
(ii) the type $\tau(H)$ matches the request $req_\alpha(\tau(G))$

(iii) α is not present in any request of $\tau(G)$

[1]Formally, the types of the AM algebra are directed acyclic graphs (Groschwitz, 2019), where source names are nodes and requests are defined by the edges. For exposition, we use a simplified definition that captures the same basic intuitions.



(a) AM term.



(b) AM dependency tree.

Figure 2.5: AM term and AM dependency tree for "The writer wants to sleep soundly" that both evaluate to the graph in Fig. 2.4a.

If one of the conditions is violated, we say that the operation is not well-typed and therefore undefined.

The last condition is important to guarantee lexically triggered re-rentrancies. From a linguistic perspective, we want G_{want} to trigger the re-entrancy in Figure 2.4a. Without condition (iii), the operation $\text{APP}_s(G_{\text{want}}, G_{\text{writer}})$ would be well-typed and its evaluation result $G_{\text{want-writer}}$ would be a graph without an s-source. Consequently, when we then perform $\text{APP}_o(G_{\text{want-writer}}, G_{\text{sleep-soundly}})$, the s-source of $G_{\text{sleep-soundly}}$ cannot merge with anything. This behavior is not desired and the well-typedness rules exclude such a term. Note however, that the other order of operations (first APP_o, then APP_s as shown in Figure 2.5a) is well-typed.

The $\text{MOD}_\beta(G, H)$ operation is allowed if and only if

(i) H has a β-source

(ii) $req_\beta(\tau(H)) = [\,]$

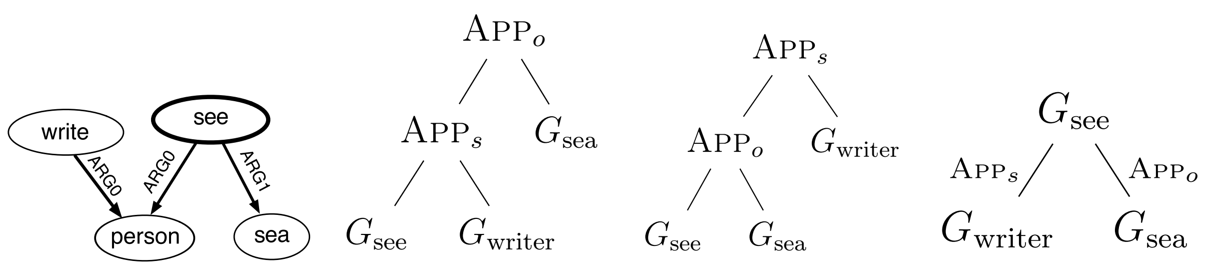
(iii) $\tau(H)$ without β is a sub-type of $\tau(G)$, written as $\tau(H) - \beta \subseteq \tau(G)$.

We say type τ is a sub-type of type τ' if all source names in τ are also in τ' and every annotation in τ is a sub-type of the respective annotation in τ'. Condition (iii) ensures that H does not introduce any sources to G that were not present already. If that were possible, that would mean that (linguistic) modifiers could change the valency of their heads, which does not seem to capture their behavior appropriately.

Importantly, we can check if an operation is allowed solely based on the types of the graphs involved without having to consider the graphs themselves. This abstraction will make it easier to reason about well-typedness later. In slight abuse of notation, we also write down AM terms on types only, e.g. $\text{APP}_s([s, o], [\,]) = [o]$.

2.2.1 AM dependency trees

There are often multiple AM terms that evaluate to the same graph and all those AM terms use the same graph constants and same operations – but in a different order. For example, the AM terms in Figure 2.6b and 2.6c both evaluate to the same graph (in Figure 2.6a). These different AM terms do not correspond to a linguistically meaningful difference and introduce a lot of spurious ambiguity, which makes the development of a parser challenging. Fortunately, there is a compact representation of these equivalent AM terms: the AM dependency tree. We will use the following definition:



(a) AMR graph. (b) (c) (d) AM dependency tree.

Figure 2.6: (a) AMR graph for "the writer sees the sea", (b) and (c) AM terms that evaluated to (a), (d) AM dependency tree.

Algorithm 1 Transformation of AM terms to AM dependency trees (Lindemann, 2018)

```
1: function TRANSFORM(AM term $t$)
2:     if $t$ has no children then
3:         return $\langle t, \emptyset \rangle$
4:     else $t$ has children $l, r$; the label at the root of $t$ is $o$
5:         let $\langle N, C \rangle = \text{TRANSFORM}(l)$
6:         return $\langle N, C \cup \{\langle \text{TRANSFORM}(r), o \rangle\} \rangle$
7:     end if
8: end function
```

Definition 2.2.1 (AM dependency tree, Lindemann (2018)). An AM dependency tree t is a tuple $\langle N, C \rangle$, where N is a constant of the AM algebra and C is a set consisting of tuples $\langle c, o \rangle$ where c is an AM dependency tree and o is a symbol of rank two of the AM algebra.

Algorithm 1 defines how we can obtain an AM dependency tree from an AM term; it returns the same AM dependency tree for all AM terms that are variations of one another because of different operation orders. For example, the AM terms in Figure 2.6b and 2.6c will both be transformed into the AM dependency tree in Figure 2.6d.

We call an AM dependency tree t well-typed if there is a well-typed AM term t' such that $\text{TRANSFORM}(t') = t$.

While an AM dependency tree represents a set of AM terms, it can be proven that all well-typed AM terms that belong to the same (well-typed) AM dependency tree evaluate to the same graph (Groschwitz, 2019). In other words, a well-typed AM dependency tree uniquely specifies a graph.

In practice, we are not only concerned with AM dependency trees but also with sentences and their relationship to one another. We usually consider only aligned AM dependency trees, that is an AM dependency together with a sentence and an injective function that maps each constant symbol to a token in the sentence. An example is shown in Figure 2.5b.

2.2.2 AM dependency parsing

The task of semantic parsing, i.e. finding a semantic graph for a given sentence, can now be seen as finding the highest scoring well-typed AM dependency for a given sentence under a scoring model, which we can then evaluate to get the graph. Groschwitz et al. (2018) score an AM dependency tree t as follows:

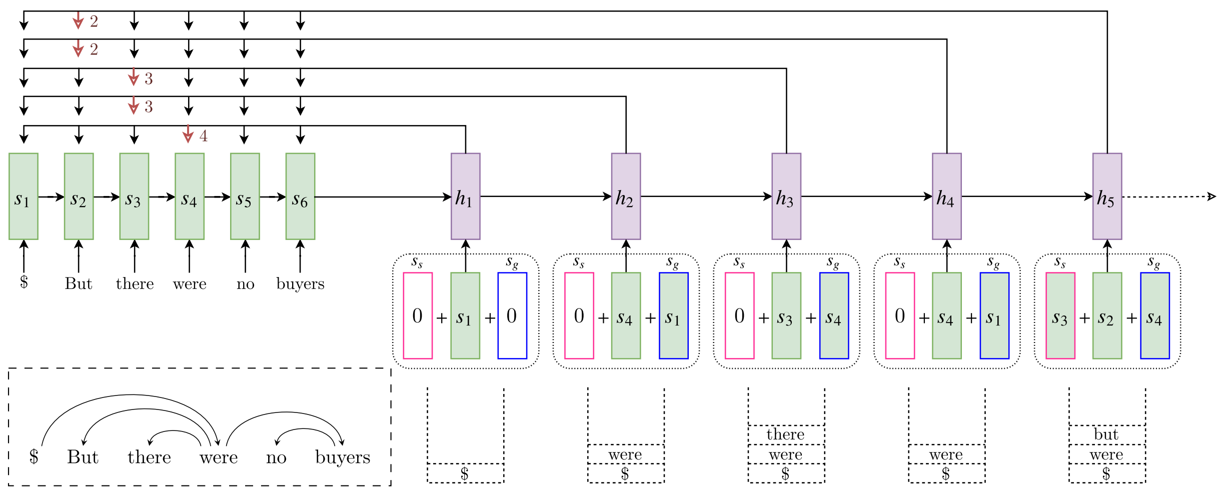


Figure 2.7: Neural top-down parsing with stack-pointer networks, copied from Ma et al. (2018). The transition sequence shown is SELECT(4), SELECT(3), POP, SELECT(2), POP.

$$\omega(t) = \sum_{i=1}^{n} \omega(G[i]) + \sum_{i \xrightarrow{l} j \in E} \omega(i \to j) + \omega(l|i \to j)$$

where $\omega(G[i])$ assigns a score to the graph constant used at token i, $\omega(i \to j)$ scores the edge from i to j and $\omega(l|i \to j)$ scores the edge label on that edge. Groschwitz et al. (2018) show that even under this simple model the problem of finding the highest scoring well-typed AM dependency tree is NP-complete. Instead, they propose two approximate algorithms:

Projective Parser As the name suggests, the projective parser restricts the search space to well-typed AM dependency trees that are projective with respect to the input sentence. The algorithm can then find the best tree in $O(n^5)$ time, where n is the length of the sentence.

Fixed tree decoder This parser first finds the best unlabeled tree first, which may or may not be projective. In a second step, it finds the highest scoring assignment of graph constants to the tokens and labels to the edges such that the AM dependency tree becomes well-typed. The run time of the second step is $O(n \cdot d^2 \cdot d)$, where d is the maximal out-degree in the unlabeled tree.

Groschwitz et al. (2018) phrase the scoring as supertagging (scoring graph constants) and dependency parsing and use a neural biLSTM model similar to that of Kiperwasser and Goldberg (2016).

Scoring graph constants for tokens can be tricky because the lexicon of graph constants becomes very large. However, there is often structure that can be exploited. For example, all transitive verbs share the same graph structure – they only differ in what we call the *lexical label*. To make the scoring task easier, Groschwitz et al. (2018) split it into scoring the the lexical label and the delexicalized graph constant, i.e. a graph constant where the lexical label is replaced by a placeholder that indicates the place where the lexical label belongs.

2.3 Top-down dependency parsing

Ma et al. (2018) proposed a top-down transition system for non-projective dependency parsing that gives very competitive performance and currently is among the state-of-the-art models. The basic idea of their approach is to build the tree from the root to the leaves in a depth-first traversal, for which they maintain a stack containing the nodes that potentially still need children and a set D of tokens that are fully processed ("done"). There are two kinds of parsing transitions:

Select(ℓ, i) A token $i \notin D$ is selected and an edge is created from the token on top of the stack to i and labeled with l. Subsequently, i is pushed to the stack and added to D.

Pop The node on top of the stack is removed. As a consequence, this node cannot get any additional children. Additionally, the node cannot be selected again.

Parsing starts with an artificial root token on top of the stack and is finished when the stack is empty and all tokens are fully processed ($\forall i . i \in D$). The parsing problem then becomes to predict a sequence of parsing transitions $d^{(1)}, \ldots, d^{(N)}$ for a given sentence $\mathbf{x}$.

$$P_\theta(d^{(1)}, \ldots, d^{(N)} | \mathbf{x}) = \prod_{t=1}^{N} P_\theta(d^{(t)} | d^{(<t)}, \mathbf{x})$$

As is usual in transition-based parsing (Nivre et al., 2006; Kiperwasser and Goldberg, 2016; Andor et al., 2016), they use a strong probability model that in fact makes no independence assumptions and encodes the entire history of transitions $d^{(<t)}$ into a vector representation $\mathbf{h}^{(t)}$. Then they model the probabilities of the next transition $d^{(t)}$ as follows:

$$\begin{aligned} &P_\theta(d^{(t)} = \textsc{Select}(\ell, j) | \mathbf{h}^{(t)}) = a_j^{(t)} \cdot P_\theta(\ell | \mathbf{h}^{(t)}, tos \to j) \\ &P_\theta(d^{(t)} = \textsc{Pop} | \mathbf{h}^{(t)}) = a_{tos}^{(t)} \end{aligned}$$

where tos refers to the token on top of the stack at step t. $a^{(t)}$ is an attention score over the tokens in the sentence. Note that the Pop can be thought of as selecting the node that is already on top of the stack.

The vector $\mathbf{h}^{(t)}$, the attention scores $a^{(t)}$ and the probabilities for edge labels are computed with a variant of pointer networks (Vinyals et al., 2015). A token x_i is represented as dense vector $\mathbf{x}'_i$ as follows:

$$\mathbf{x}'_i = [E(x_i), E(p_i), \mathrm{CharCNN}(x_i)]$$

where p_i is the POS tag of x_i and E is an embedding function. CharCNN is a convolutional neural network (CNN) that operates on character-basis and thus can take morphological information into account. Then, a multi-layer bidirectional LSTM is run on $\mathbf{x}'_1, \ldots, \mathbf{x}'_n$ to produce a sequence of states $\mathbf{s}_1, \ldots, \mathbf{s}_n$ that represent the respective tokens and their syntactic context. The last hidden state of each direction is used to initialize a decoder LSTM which is updated as follows

$$\mathbf{h}^{(t)} = \mathrm{LSTM}(\mathbf{h}^{(t-1)}, \mathbf{s}_{tos} + \mathbf{s}_p + \mathbf{s}_c)$$

where again p refers to the parent of the token on top of the stack (tos) and c the most recently generated child of tos. If such nodes do not exist then $\mathbf{s}_p$ and $\mathbf{s}_c$ are treated as vectors of zeros. Thus, for each decision, the parser particularly takes into account the active node on top of the stack and a bit of syntactic context in the shape of the most recently generated child and the parent node. Access to the rest of the tree-structure is provided through the LSTM.

The attention values follow a bi-affine scoring mechanism with MLPs that "strip information that is irrelevant for this decision", popularized by Dozat and Manning (2017):

$$\begin{aligned}
\mathbf{g}^{(t)} &= \mathrm{MLP}^{head}(\mathbf{h}^{(t)}) \\
\mathbf{r}_j &= \mathrm{MLP}^{dep}(\mathbf{s}_j) \\
e_j^{(t)} &= \mathbf{r}_j^T \mathbf{W}\mathbf{g}^{(t)} + \mathbf{U}\mathbf{g}^{(t)} + \mathbf{V}\mathbf{r}_j + \mathbf{b} \\
a_i^{(t)} &= \mathrm{softmax}(e^{(t)})_i \qquad (2.1)
\end{aligned}$$

where $\mathbf{W}, \mathbf{U}, \mathbf{V}$ are weight matrices and $\mathbf{b}$ is a bias. They use an analogous scoring mechanism to compute $P_\theta(\ell|\mathbf{h}^{(t)}, tos \to j)$, where again $\mathbf{h}^{(t)}$ and $\mathbf{s}_j$ are first projected into another space with MLPs and then all labels are scored with a bi-affine transformation.

Figure 2.7 shows the architecture and a parsing example for the sentence "But there were no buyers". The \$ symbol is an artificial root node that is on the stack in the initial configuration of the parser.

Since they encode the entire history of parsing transitions, exact inference is intractable, and they perform greedy search or beam search. At test time, they also use a constrained model that only allows to SELECT tokens that have not been selected yet. In particular, this means that they set

$$P_\theta(d^{t+1} = \textsc{Select}(\ell, j)|\mathbf{h}^{(t)}) = \frac{1}{Z}[\![j \notin D]\!] a_j^{(t)} \cdot P_\theta(\ell|\mathbf{h}^{(t)}, tos \to j)$$

where $[\![j \notin D]\!]$ is 1 if $j \notin D$ and 0 otherwise. D contains all tokens that have been selected before. Z is a constant for re-normalization. Note that there is no need to actually compute Z because we are only interested in the most likely valid sequence of transitions and that does not change by multiplying all valid sequences with a constant $Z > 0$.

The model is trained with teacher forcing, that is, during training the model is exposed to a prefix of the gold action sequence and has to predict the next action. The sequence of gold transitions can be pre-computed, which allows them to also pre-compute which tokens are on top of the stack at any point in time, which makes batching easy and training efficient.

Note that the order in which the children are selected has no influence on the final tree (in Figure 2.7, "but" could be selected before "there") and thus there is usually more than one gold transition sequence. Ma et al. deal with this by fixing the order a-priori and find that an inside-out order works best (also shown in Figure 2.7).

The run time complexity of inference is $O(n^2)$ with n being the sentence length because there is a linear number of transitions and every transition requires computing attention scores to all n tokens in the sentence.

Chapter 3

Transition systems

In order for an AM dependency parser to be useful in practice, it should have the following properties:

(i) Soundness: all derived AM dependency trees are well-typed.

(ii) Completeness: all well-typed trees can be derived.

(iii) Good worst-case run time in the length of the sentence.

(iv) Fast in practice.

Good worst-case run times make transition systems a plausible choice, which is the approach pursued in this thesis. A natural starting point are transition systems for regular dependency parsing. The most common approaches like arc-standard (Nivre, 2003) maintain a stack and process the sentence from left to right. Unfortunately, the order in which the tree is built in these systems makes it difficult to guarantee well-typedness. In particular, if several subtrees are on the stack, it is not clear that there is always a way to combine them. If there is no way of combining partial results in a well-typed way, the transition system runs into a dead end and has to backtrack. Backtracking is undesirable because it leads to high worst-case run times.

In this chapter, we will introduce transition systems for AM dependency parsing. All transition systems proceed top-down and recursively build the AM dependency tree. The first transition system, lexical type first (LTF) (Section 3.2), selects the graph constant of a node *before* drawing any outgoing edges. We first consider a naive version that can run into dead ends and then revise it to exclude dead ends. In Section 3.3, we present a transition system where the outgoing edges of a node are drawn first; only then we choose the graph constant. This transition system is called lexical type last (LTL).

In Section 3.4, we prove that the transition systems are sound and complete. Making a few mild assumptions about the graph lexicon and the edge labels at our disposal, we can also show that they cannot run into dead ends.

Throughout this chapter, we assume that we are given a set of graph constants C that can draw source names from a fixed finite set $\mathcal{S}$, a set of types Ω and a set of edge labels *Lab* with App and mod labels and a special *root* label.

3.1 Term types and apply sets

Before we describe the parsing algorithms, we introduce the notions of term type and apply set.

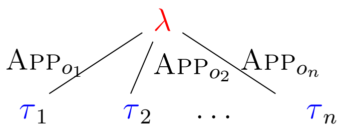


Figure 3.1: AM dependency tree with types as nodes.

In an AM dependency tree, there are two kinds of types for any node: the type of the graph constant, which is called lexical type (often written λ), and the type of the subtree τ, called term type, which is defined below:

Definition 3.1.1 (Term type). Let t be a well-typed AM dependency tree with i being a node of t. The term type τ at i is the type of the graph that the subtree rooted in i evaluates to.

Note that there is no inherent difference in the types themselves that make them a lexical type or a term type; the distinction is made relative to a particular AM dependency tree. Examples of lexical types and term types are shown below the words in Figure 3.3. Note that term type and lexical type coincide for leaves in any AM dependency tree.

The basic idea for the transition systems presented here is to recursively build subtrees of a specified term type. For the AM dependency tree representing the entire sentence, it is the empty type, and for all its subtrees, the term type is a consequence of the lexical type of the parent and the label of the incoming edge.

Definition 3.1.2 (Apply set). The apply set $\mathcal{A}(\lambda, \tau)$ is the set O of n source names such that there exists a sequence $o_1, \dots, o_n \in O$ with

$$\text{App}_{o_n}(\dots \text{App}_{o_2}(\text{App}_{o_1}(\lambda, \tau_1), \tau_2), \dots \tau_n) = \tau$$

for some types $\tau_1, \dots, \tau_n$.

For example, the apply set from $[s, o]$ to $[\,]$ is $\mathcal{A}([s, o], [\,]) = \{s, o\}$ and from $[s, o[s]]$ to $[s]$ is $\mathcal{A}([s, o[s]], [s]) = \{o\}$. The apply set from $[\,]$ to $[s, o]$ is not defined because there is *no* sequence of the form $\text{App}_{o_n}(\dots \text{App}_{o_2}(\text{App}_{o_1}([\,], \tau_1), \tau_2), \dots \tau_n) = [s, o]$ because App operations cannot introduce sources.

We can also think of the AM term in the definition as an AM dependency tree, which is shown in Figure 3.1. If we know that a certain node i has a graph constant G, and we know that i must have a specific term type τ, the apply set $\mathcal{A}(\lambda, \tau)$ tells us exactly what outgoing App edges i must have to ensure well-typedness. If we want to actually draw those edges, the apply set also gives us the freedom to choose whatever order we want as long as we commit to drawing all outgoing App edges that correspond to the apply set.

The definition of the apply set does not immediately tell us how it can be computed; we treat its computation as a black box here and only note that it can be done efficiently[1].

Sometimes we only want to know if an apply set exists and are not so much interested in what it is:

Definition 3.1.3 (Apply reachable). The type τ is apply reachable from type λ iff $\mathcal{A}(\lambda, \tau)$ is defined.

[1]If the apply set $\mathcal{A}(\lambda, \tau)$ exists, it contains those source names that are in λ but not in τ. Let us call this set A. To check if the apply set indeed exists, we have to check if we can create a sequence of App operations that is well-typed and that uses exactly those source names in A. This can be done by using the fact that types are formally directed, acyclic graphs (Groschwitz, 2019): we have to check if we can obtain τ by iteratively removing a node $n \in A$ without any incoming edge from λ.

3.2 Lexical type first

3.2.1 Naive approach

We are now ready to define a first version of the transition system for our parser. The parser builds a dependency tree top-down and manipulates *parser configurations* to track parsing decisions and ensure well-typedness.

A parser configuration $\langle \mathbb{E}, \mathbb{T}, \mathbb{A}, \mathbb{G}, \mathbb{S} \rangle$ consists of four partial functions $\mathbb{E}, \mathbb{T}, \mathbb{A}, \mathbb{G}$ that map each token i to, respectively: the labeled incoming edge of i, written $j \xrightarrow{\ell} i$; the set of possible term types at i; the sources of outgoing App edges at i, i.e. which sources of the apply set we have covered; and the graph constant at i. These functions are partial, i.e. they may be undefined for some nodes. $\mathbb{S}$ is a stack of nodes that potentially still need children; we call the node on top of $\mathbb{S}$ the *active node*.

The initial configuration is $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. A *goal configuration* has an empty stack and for all tokens i, it holds either that i is ignored and thus has no incoming edge, or that for some type τ and graph G, $\mathbb{T}(i) = \{\tau\}$, $\mathbb{G}(i) = G$ and $\mathbb{A}(i) = \mathcal{A}(\tau(G), \tau)$. The latter condition (for some type τ and graph G, $\mathbb{T}(i) = \{\tau\}$, $\mathbb{G}(i) = G$) must hold for at least one token i.

The transition rules below read as follows: everything above the line denotes preconditions on when the transition can be applied; for example, that $\mathbb{T}$ must map node i to some set T of types. The transition rule then updates the configuration by adding what is specified below the line.

An example run is shown in Figure 3.2.

Init An Init(i) transition is always the first transition and makes i the root of the tree and pushes i to the stack:

$$\begin{array}{lllll} \mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \hline 0 \xrightarrow{root} i & i \mapsto \{[\,]\} & & & \mathrm{i} \end{array}$$

Fixing the term type as $[\,]$ ensures that the overall evaluation result has no unfilled sources left.

Choose If we have not yet chosen a graph constant for the active node, we assign one with the Choose(τ, G) transition:

$$\begin{array}{lllll} \mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\ & i \mapsto T & & i \notin Dom(\mathbb{G}) & \sigma|i \\ \hline & i \mapsto \{\tau\} & i \mapsto \emptyset & i \mapsto G & \sigma|i \end{array}$$

Here $Dom(\mathbb{G})$ refers to the domain of $\mathbb{G}$, that is the elements for which $\mathbb{G}$ is defined. This transition may only be applied if the specific term type $\tau \in T$ is apply reachable from the newly selected lexical type $\tau(G)$. Choose determines the lexical type of i *first*, before any outgoing edges are added.

We define a helper function $PossT$ as follows:

$$PossT(\lambda, \ell) = \begin{cases} \{req_\alpha(\lambda)\} & \text{if } \ell = \mathrm{App}_\alpha \\ \{\tau \in \Omega \mid \tau - \beta \subseteq \lambda \wedge req_\beta(\tau) = [\,]\} & \text{if } \ell = \mathrm{MOD}_\beta \end{cases}$$

PossT returns for a lexical type λ and an edge label ℓ a set of term types T and mirrors the well-typedness conditions in Section 2.2 such that $\ell(\lambda, \tau)$ is a well-typed for all $\tau \in T$.

Apply Once the term type τ and graph G of the active node i have been chosen, the $\text{APPLY}(\alpha, j)$ operation can draw an $\text{APP}_\alpha \in \mathit{Lab}$ edge to a node j that has no incoming edge, adding j to the stack:

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
j \notin \mathit{Dom}(\mathbb{E}) & i \mapsto \{\tau\} & i \mapsto A, \alpha \notin A, \alpha \in \mathcal{A}(\tau(G), \tau) & i \mapsto G & \sigma|i \\
\hline
i \xrightarrow{\text{APP}_\alpha} j & j \mapsto \mathit{PossT}(\tau(G), \text{APP}_\alpha) & i \mapsto A \cup \{\alpha\} & & \sigma|i|j
\end{array}$$

Here α must be a source in the apply set $\mathcal{A}(\tau(G), \tau)$ but not in $\mathbb{A}(i)$, i.e. it must be a source of G that still needs to be filled. Fixing the term type of j ensures the type restrictions of the APP_α operation.

Modify In contrast to outgoing APP edges, which are determined by the apply set, we can add arbitrary outgoing MOD edges to the active node i. This is done with the transition $\text{MODIFY}(\beta, j)$, which draws a $\text{MOD}_\beta \in \mathit{Lab}$ edge to a token j without an incoming edge, pushing j onto the stack:

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
j \notin \mathit{Dom}(\mathbb{E}) & i \mapsto \{\tau\} & i \mapsto A & i \mapsto G & \sigma|i \\
\hline
i \xrightarrow{\text{MOD}_\beta} j & j \mapsto \mathit{PossT}(\tau(G), \text{MOD}_\beta) & & & \sigma|i|j
\end{array}$$

Here we set the term type of j to be $\mathit{PossT}(\tau(G), \text{MOD}_\beta)$ which ensures well-typedness.

Pop The POP transition decides that an active node that has all of its APP edges will not receive any further outgoing edges, and removes it from the stack.

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
 & i \mapsto \{\tau\} & i \mapsto \mathcal{A}(\tau(G), \tau) & i \mapsto G & \sigma|i \\
\hline
 & & & & \sigma
\end{array}$$

The important restriction about this rule is that $\mathbb{A}_c(i) = \mathcal{A}(\tau(G), \tau)$ because this means that we have created *all* APP edges that we need.

Example We illustrate the transition rules with the derivation in Figure 3.2, which derives the AM dependency tree in Figure 3.3.

In the first step, we can only apply one of the INIT transitions. Here, we decide for INIT(3), making *wants* the root of the dependency tree, specifying that its term type has to be the empty type and pushing it to the stack.

We then make a CHOOSE transition, which assigns a term type τ and a graph G to the token on top of the stack, *wants*. The term type must be $[\,]$, because this is the only element in the term-type set specified by INIT. In the CHOOSE transition, we assign the token *wants* the graph constant G_{want}, and thus the lexical type $[s, o[s]]$. This is allowed because the term type $[\,]$ is apply reachable from $[s, o[s]]$, with the apply set $\mathcal{A}([s, o[s]], [\,]) = \{s, o\}$. We note down that we have not covered any part of the apply set.

Step	$\mathbb{E}$	$\mathbb{T}$	$\mathbb{A}$	$\mathbb{G}$	$\mathbb{S}$	Transition
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$[]$	
2	$0 \xrightarrow{\text{ROOT}} wants_3$	$wants_3 \mapsto \{[\,]\}$			3	$\text{INIT}(3)$
3			$wants_3 \mapsto \emptyset$	$wants_3 \mapsto G_{\text{want}}$	3	CHOOSE $([\,], G_{\text{want}})$
4	$wants_3 \xrightarrow{\text{APP}_s} writer_2$	$writer_2 \mapsto \{[\,]\}$	$wants_3 \mapsto \{s\}$		3 2	APPLY (s, 2)
5			$writer_2 \mapsto \emptyset$	$writer_2 \mapsto G_{\text{writer}}$	3 2	CHOOSE $([\,], G_{\text{writer}})$
6					3	POP
7	$wants_3 \xrightarrow{\text{APP}_o} sleep_5$	$sleep_5 \mapsto \{[s]\}$	$wants_3 \mapsto \{s, o\}$		3 5	APPLY (o, 5)
8			$sleep_5 \mapsto \emptyset$	$sleep_5 \mapsto G_{\text{sleep}}$	3 5	CHOOSE $([s], G_{\text{sleep}})$
9	$sleep_5 \xrightarrow{\text{MOD}_m} soundly_6$	$soundly_6 \mapsto \{[m], [s, m]\}$			3 5 6	MODIFY (m, 6)
10		$soundly_6 \mapsto \{[m]\}$	$soundly_6 \mapsto \emptyset$	$soundly_6 \mapsto G_{\text{soundly}}$	3 5 6	CHOOSE $([m], G_{\text{soundly}})$
11					$[]$	3 × POP

Figure 3.2: Derivation with LTF of the AM dependency tree in Fig. 3.3. The steps show only what changed for $\mathbb{E}, \mathbb{T}, \mathbb{A}$ and $\mathbb{G}$; the stack $\mathbb{S}$ is shown in full.

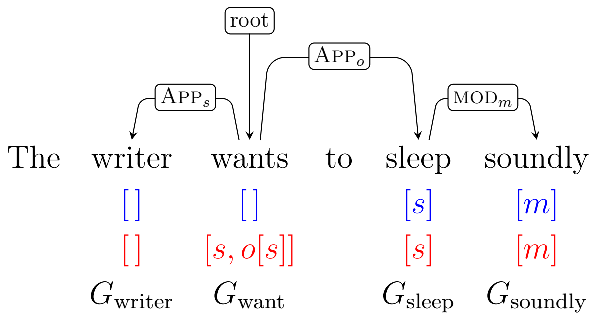


Figure 3.3: An AM dependency aligned to a sentence. Below the words with incoming edges are: the term type, the lexical type and the graph constant symbol.

In step 4, we fill the s-source in this apply set using an APPLY$(s, 2)$ transition. This creates an APP$_s$-edge from the token on top of the stack, *wants*, to *writer*. APPLY pushes *writer* on top of the stack, and we determine its term type to be $[\,]$, matching the request of $[s, o[s]]$ at s. We also note down in $\mathbb{A}(3)$ that s has been filled.

In step 5, we CHOOSE the lexical as-graph of *writer* with type $[\,]$. Since term type and lexical type of *writer* are identical, the apply set is empty, and we can POP it off the stack immediately.

In step 7, we use an APPLY$(o, 5)$ transition to create the APP$_o$ edge from 3 (top of the stack) to 5. The transition can be applied because $o \in \mathcal{A}([s, o[s]], [\,]) = \{s, o\}$. The transition also pushes token 4, *sleep*, to the stack and determines its term type to be $[s]$ to match its the request of $[s, o[s]]$ at o.

In step 8, we have to choose a graph constant for *sleep*. The term type is already determined from step 7.

In step 9, we apply a MODIFY transition to create a MOD$_m$ edge from *sleep* to *soundly*. Since the token on top of the stack, *sleep*, has lexical type $[s]$, both $[m]$ and $[s, m]$ are valid term types for *soundly*. In the subsequent CHOOSE transition we pick $[m]$ as the term type of *soundly* and G_{soundly} to be the lexical as-graph with lexical type $[m]$. Since the apply set $\mathcal{A}([m], [m]) = \emptyset$, *soundly* cannot get outgoing APP edges. While the transition system would allow a MOD edge at this point, we decide to POP in step 11. Since we neither want to add an outgoing MOD edge from *sleep* nor from *wants*, we POP again twice.

Finally, note that the way the transition system is set up, it builds the tree in a depth-first manner.

3.2.2 Without dead ends

While the above parser guarantees well-typedness when it completes, it can still get stuck. This is because when we CHOOSE a term type τ and lexical type λ for a node, we *must* perform APPLY transitions for all sources in their apply set $\mathcal{A}(\lambda, \tau)$ to reach a goal configuration. But every APPLY transition adds an incoming edge to a token that did not have one before; if our choices for lexical and term types require more APPLY transitions than there are tokens without incoming edge, the parser cannot reach a goal configuration.

To avoid this situation, we track for each configuration c the difference $W_c - O_c$ of the number W_c of tokens without an incoming edge and the number O_c of APPLY transitions we 'owe' to fill all sources. O_c is obtained by summing across all tokens i the number $O_c(i)$ of APP children i still needs. To generalize to cases where we may not yet know the graph constant for i, we let $K_c(i) = \{\tau(\mathbb{G}_c(i))\}$ if $i \in \mathit{Dom}(\mathbb{G}_c)$ and $K_c(i) = \Omega$ otherwise. Then we can define

$$O_c(i) = \min_{\lambda \in K_c(i), \tau \in \mathbb{T}_c(i)} |\mathcal{A}(\lambda, \tau) - \mathbb{A}_c(i)|.$$

If $\mathbb{T}$ or $\mathbb{A}$ is not defined for i, we let $O_c(i) = 0$.

Finally, given a type τ, an upper bound n, and a set A of already-covered sources, we let $\mathit{PossL}(\tau, A, n)$ be the set of lexical types λ such that $A \subseteq \mathcal{A}(\lambda, \tau)$ and we can reach τ from λ with APP operations for the sources in A and at most n additional APP operations:

$$\begin{aligned} \mathit{PossL}(\tau, A, n) = \{\lambda \in \Omega \mid A \subseteq \mathcal{A}(\lambda, \tau) \wedge \\ |\mathcal{A}(\lambda, \tau) - A| \leq n\} \end{aligned}$$

We prevent dead ends in LTF by making the following two restrictions:

Revised CHOOSE CHOOSE(τ, G) can only be applied to a configuration c if in addition to the other constraints $\tau(G) \in \mathit{PossL}(\tau, \emptyset, W_c - O_c)$. Then τ is apply reachable from $\tau(G)$ with at most $W_c - O_c$ APPLY transitions; this is exactly as many as we can spare.

Revised MODIFY The MODIFY transition reduces the number of tokens that have no incoming edge without performing a APPLY transition, so we only allow it when we have tokens 'to spare', i.e. $W_c - O_c \geq 1$.

The example in Figure 3.2 works analogously in the revised version of LTF without dead ends; the only differences are to check the additional constraints, which are all met in this case.

3.2.3 Shorter transition sequences

To obtain high parsing speed in practice, it is not only important that a sentence of length n can be parsed with a transition sequence of length $O(n)$, but also what the constant factor is. For every token with an incoming edge, there are three transitions in LTF that have to be performed: the transition to put it on the stack (INIT or APPLY or MODIFY), CHOOSE and POP. That is, for a sentence of length n, there are at most $3n$ transitions, depending on how many tokens are ignored.

We will define a probability model (Chapter 4) and use a decoder network that makes predictions for edges, edge labels and graph constants at every step, no matter which transition we will finally choose. If we can re-arrange the transitions such that they do the

Transitions	**Edge**	**Graph constant**
$\textsc{Init}(i)$	✓	×
$\textsc{Apply}(\alpha, j)$	✓	×
$\textsc{Modify}(\beta, j)$	✓	×
$\textsc{Pop}$	×	×
$\textsc{Choose}(\tau, G)$, $\textsc{Apply}(\alpha, j)$	✓	✓
$\textsc{Choose}(\tau, G)$, $\textsc{Modify}(\beta, j)$	✓	✓
$\textsc{Choose}(\tau, G)$, $\textsc{Pop}$	×	✓

Figure 3.4: LTF transition system with combined transitions. The last two columns show contribution of transition to AM dependency tree.

Step	**Transitions**
1	$\textsc{Init}(3)$
2	$\textsc{Choose}([\,], G_{\text{want}})$, $\textsc{Apply}(s, 2)$
3	$\textsc{Choose}([\,], G_{\text{writer}}), \textsc{Pop}$
4	$\textsc{Apply}(o, 5)$
5	$\textsc{Choose}([s], G_{\text{sleep}}), \textsc{Modify}(m, 6)$
6	$\textsc{Choose}([m], G_{\text{soundly}}), \textsc{Pop}$
7	$\textsc{Pop}$
8	$\textsc{Pop}$

Figure 3.5: Transition sequence with *LTF with combined transitions* for example in Figure 3.2.

same work but make the transition sequences shorter, we need fewer steps in the decoder network and that saves time.

If we look at our transitions, we see that $\textsc{Choose}(\tau, G)$ is the only transition that predicts graph constants and term types but does not add a new child or new label to the AM dependency tree. This means we do not make efficient use of the possibility to predict a graph constant at each step. That is, whenever we use a transition that is not $\textsc{Choose}(\tau, G)$ we potentially lose an opportunity to choose a graph constant.

To overcome this, we can group a $\textsc{Choose}(\tau, G)$ transition with the subsequent one to a *combined transition*. Figure 3.4 shows the transitions of *LTF with combined transitions* which makes better use of the possibility to predict a graph constant and an edge at every point in time. A combined transition can be applied if and only if $\textsc{Choose}(\tau, G)$ is applicable and the subsequent transition is applicable if we take the effect of $\textsc{Choose}(\tau, G)$ into account. We can translate a transition sequence from LTF to LTF with combined transitions by going over it from the beginning to the end and combining the current transition with the subsequent one whenever possible and otherwise choosing the original one. Note that this transformation is a bijection so every result about LTF can be transferred to LTF with combined transitions.

Since $\frac{1}{3}$ of the performed transitions in LTF are $\textsc{Choose}(\tau, G)$ transitions, grouping it with the subsequent one allows us to reduce the length of the transition sequences to $2n$, which translates to fewer steps with the decoder network.

Figure 3.5 shows the transition sequence of LTF with combined transitions for the example originally presented in Figure 3.2.

3.3 Lexical type last

The lexical type first transition system chooses the graph constant for a token early, and then chooses outgoing APP edges that fit the lexical type. But of course the decisions on lexical type and outgoing edges interact. Thus we also consider a transition system in which the lexical type is chosen *after* deciding on the outgoing edges.

INIT The INIT transition for LTL is almost the same as that for LTF. The difference is that we initialize $\mathbb{A}$ so that we can immediately draw outgoing APP edges:

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\hline
0 \xrightarrow{root} i & i \mapsto \{[\,]\} & i \mapsto \emptyset & & \mathrm{i}
\end{array}$$

APPLY In contrast to LTF, APPLY and MODIFY do not assign term types to children and do not push the children to the stack. This allows the transition system to add outgoing edges to the active node i without committing to types. The $\text{APPLY}(\alpha, j)$ creates the edge $\text{APP}_\alpha \in \mathit{Lab}$ from the active node i to j and looks as follows:

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
j \notin \mathit{Dom}(\mathbb{E}) & i \mapsto T & i \mapsto A, \alpha \notin A & & \sigma|i \\
\hline
i \xrightarrow{\text{APP}_\alpha} j & & i \mapsto A \cup \{\alpha\} & & \sigma|i
\end{array}$$

We check if there are *possible* types τ and λ with α in their apply set $\mathcal{A}(\lambda, \tau)$, by checking that $\bigcup_{\tau \in T} \mathit{PossL}(\tau, A \cup \{\alpha\}, W_c - 1)$ is non-empty. Otherwise, the transition cannot be applied. This ensures that we can always find a viable lexical type for i after drawing at most $W_c - 1$ additional APP edges. We also keep the restriction that $\alpha \notin A$, to avoid duplicate APP_α edges.

MODIFY $\text{MODIFY}(\beta, j)$ only creates the edge $\text{MOD}_\beta \in \mathit{Lab}$ from the active node i to j without any other effects:

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
j \notin \mathit{Dom}(\mathbb{E}) & & & & \sigma|i \\
\hline
i \xrightarrow{\text{MOD}_\beta} j & & & & \sigma|i
\end{array}$$

The MODIFY transition reduces the number of tokens that have no incoming edge without performing a APPLY transition, so we only allow it when we have tokens 'to spare', i.e. $W_c - O_c \geq 1$.

FINISH We then replace CHOOSE and POP with a single transition $\text{FINISH}(G)$, which selects an appropriate graph constant G for the active node i and pops i off the stack, such that no more edges can be added.

$$\begin{array}{lllll}
\mathbb{E} & \mathbb{T} & \mathbb{A} & \mathbb{G} & \mathbb{S} \\
i \xrightarrow{\text{APP}_{\alpha_k}} j_k & & & & \\
i \xrightarrow{\text{MOD}_{\beta_k}} m_k & i \mapsto T & i \mapsto A & & \sigma|i \\
\hline
& i \mapsto \{\tau\}, & & i \mapsto G & \sigma|m_1|\ldots|m_r \\
& j_k \mapsto \mathit{PossT}(\tau(G), \text{APP}_{\alpha_k}), & j_k \mapsto \emptyset, & & |j_1|\ldots|j_s \\
& m_k \mapsto \mathit{PossT}(\tau(G), \text{MOD}_{\beta_k}) & m_k \mapsto \emptyset & &
\end{array}$$

Step	$\mathbb{E}$	$\mathbb{T}$	$\mathbb{A}$	$\mathbb{G}$	$\mathbb{S}$	Transition
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	[]	
2	$0 \xrightarrow{\text{ROOT}} \mathit{wants}_3$	$\mathit{wants}_3 \mapsto \{[\,]\}$	$\mathit{wants}_3 \mapsto \emptyset$		3	Init 3
3	$\mathit{wants}_3 \xrightarrow{\text{App}_s} \mathit{writer}_2$		$\mathit{wants}_3 \mapsto \{s\}$		3	Apply (s, 2)
4	$\mathit{wants}_3 \xrightarrow{\text{App}_o} \mathit{sleep}_5$		$\mathit{wants}_3 \mapsto \{s, o\}$		3	Apply (o, 5)
5		$\mathit{writer}_2 \mapsto \{[\,]\}$, $\mathit{sleep}_5 \mapsto \{[s]\}$	$\mathit{writer}_2 \mapsto \emptyset$, $\mathit{sleep}_5 \mapsto \emptyset$	$\mathit{wants}_3 \mapsto G_{\text{want}}$	5 2	Finish$(\langle G_{\text{want}}, [s, o[s]]\rangle)$
6				$\mathit{writer}_2 \mapsto G_{\text{writer}}$	5	Finish$(\langle G_{\text{writer}}, [\,]\rangle)$
7	$\mathit{sleep}_5 \xrightarrow{\text{MOD}_m} \mathit{soundly}_6$				5	Modify (m, 6)
8		$\mathit{soundly}_6 \mapsto \{[m], [s, m]\}$		$\mathit{sleep}_5 \mapsto G_{\text{sleep}}$	6	Finish$(\langle G_{\text{sleep}}, [s]\rangle)$
9		$\mathit{soundly}_6 \mapsto \{[m]\}$		$\mathit{soundly}_6 \mapsto G_{\text{soundly}}$		Finish$(\langle G_{\text{soundly}}, [m]\rangle)$

Figure 3.6: Derivation with LTL of the AM dependency tree in Fig. 3.3. The steps show only what changed for $\mathbb{E}, \mathbb{T}, \mathbb{A}$ and $\mathbb{G}$; the stack $\mathbb{S}$ is shown in full. The chosen graph constants are annotated with their lexical types.

Finish(G) is allowed if $\mathcal{A}(\tau(G), \tau) = A$ for some $\tau \in T$, and fixes this τ as the term type. Finish also pushes the child nodes j_k of all $s \geq 0$ outgoing App edges and the child nodes m_k of all $r \geq 0$ outgoing MOD edges onto the stack and initializes their $\mathbb{T}$-functions according to PossT so that well-typedness restrictions are met.

In practice, we push the child nodes in a systematic order to the stack, namely in the reverse order of when they are created, so that they are popped off the stack in the order the edges were drawn.

Example We consider the example in Figure 3.6, depicting a derivation of the same AM dependency tree in Figure 3.3. The Init(3) transition behaves similarly to LTF but also initializes the set $\mathbb{A}$.

In step 2, we use an Apply$(s, 2)$ transition, which creates the edge from wants_3 to writer_2. This transition can be applied because $\mathit{PossL}([\,], \{s\}, 5-1)$ is non-empty, for example, it contains $[s]$. In step 3, we create the App$_o$ edge with Apply$(o, 5)$ – again this is possible because $\mathit{PossL}([\,], \{s, o\}, 4-1)$ is non-empty; it contains $[s, o[s]]$.

In step 5, we can Finish with G_{want}, which has lexical type $[s, o[s]]$. This is possible because $\mathcal{A}([s, o[s]], [\,]) = \{s, o\} = \mathbb{A}(\mathit{wants}_3)$. Finish pops the current node off the stack and pushes the children of wants_3 in the opposite order they were attached. In step 6, writer_2 is on top of the stack, and we decide to Finish immediately with G_{writer}. We can do so, because $\mathcal{A}([\,], [\,]) = \emptyset = \mathbb{A}(\mathit{writer}_2)$. This leaves sleep_5 on the stack, from which we draw a MOD$_m$ edge to $\mathit{soundly}_6$. The transition is applicable because $W_c = 3$ and $O_c = 0$ and thus $W_c - O_c \geq 1$.

In step 8, we Finish with G_{sleep}. For its MOD$_m$ child, $\mathit{soundly}_6$, this means, it can have one of the term types $\{[m], [s, m]\}$. We leave the term type ambiguous and could add outgoing edges but in this example, we Finish right away with G_{soundly}, which has lexical type $[m]$. Note that the term type, $[m]$, follows from our choice of graph constant and the outgoing edges we have drawn (here: none).

Finally, note that the transition system does not build the tree strictly in a depth-first manner (in contrast to LTF) but rather first draws all outgoing edges of a node and then recursively continues with the children.

3.4 Theoretical guarantees

The transition systems LTF and LTL are designed in such a way that they enjoy three particularly important properties: soundness, completeness and the lack of dead ends. In this section, we phrase those guarantees in formal terms, determine which assumptions are needed and prove the guarantees.

The definition of a goal condition is quite strict but it can be shown that for LTF and LTL simpler conditions are equivalent:

Lemma 3.4.1. Let c be a configuration derived by LTF. c is a goal configuration if and only if $\mathbb{S}_c$ is empty and $\mathbb{G}_c$ is defined for some i.

Proof. $\Longrightarrow$
This follows trivially from the definition of a goal condition.
$\Longleftarrow$
We have to validate that for each token l, either l is ignored and thus has no incoming edge, or that for some type τ and graph G, $\mathbb{T}_c(l) = \{\tau\}$, $\mathbb{G}_c(l) = G$ and $\mathbb{A}_c(l) = \mathcal{A}(\tau(G), \tau)$. Additionally, there must be at least one token j such that $\mathbb{T}_c(j) = \{\tau\}$, $\mathbb{G}_c(j) = G$ and $\mathbb{A}_c(j) = \mathcal{A}(\tau(G), \tau)$. We first show that this latter condition holds for token i for which $\mathbb{G}_c$ is defined. Notice that i must have been on the stack and a CHOOSE transition has been applied. Since it is not on the stack anymore, a POP transition has been applied in some configuration c' where i was the active node. This means that $\mathbb{A}_c(i) = \mathbb{A}_{c'}(i) = \mathcal{A}(\tau(\mathbb{G}_c(i)), \tau)$ with $\mathbb{T}_c(i) = \mathbb{T}_{c'}(i) = \{\tau\}$ and thus i fulfills its part for c being a goal configuration.

We assumed that c was derived by LTF, so let s be an arbitrary transition sequence that derives c from the initial state (there might be multiple). We can divide the tokens in the sentence into two groups, based on whether they have ever been on the stack over the course of s:

- let j be an arbitrary token such that there is a state c' produced by a prefix of the transition sequence s where j is on the stack. Here, the same argument holds as above: since j is no longer on the stack, a POP transition must have been applied which implies that $\mathbb{A}_c(j) = \mathcal{A}(\tau(\mathbb{G}_c(j)), \tau)$ with $\mathbb{T}_c(j) = \{\tau\}$.

- let j be an arbitrary token such that there is *no* state c' produced by a prefix of the transition sequence s where j is on the stack. Clearly, such a token j does not have an incoming edge and thus also fulfills its part.

□

Lemma 3.4.2. Let c be a configuration derived by LTL. c is a goal configuration if and only if $\mathbb{S}_c$ is empty and $\mathbb{G}_c$ is defined for some i.

Proof. The proof is analogous to the proof of Lemma 3.4.1. □

3.4.1 Soundness

An important property of the transition systems is that they are sound, that is, every AM dependency tree they derive is well-typed.

Theorem 3.4.3 (Soundness). For every goal configuration c derived by LTF or LTL, the AM dependency tree described by c is well-typed.

Here, "described by" means that we can read off the AM dependency tree from the set of edges $\mathbb{E}_c$ and graph constants $\mathbb{G}_c$. We do not need any additional assumptions to prove this theorem.

Before we can prove the theorem we first need the following lemma:

Lemma 3.4.4. In every configuration c derived by LTF or LTL, token i has an APP_α child if and only if $\alpha \in \mathbb{A}_c(i)$.

Proof. The $\text{APPLY}(\alpha, j)$ transitions in LTF and LTL always add an α-source to $\mathbb{A}_c(i)$ and simultaneously add an APP_α edge. There are no other ways to add a source to $\mathbb{A}_c(i)$ or to create an APP_α edge. □

To prove the theorem, first observe that LTF and LTL only derive trees. Well-typedness then follows from applying the following lemma to the root of the tree in the goal configuration c:

Lemma 3.4.5. Let c be a goal configuration derived by LTF or LTL and i be a token with $\mathbb{T}_c(i) = \{\tau\}$. Then the subtree rooted in i is well-typed and has type τ.

Proof. By structural induction over the subtrees.

Base case Since i has no children, it has no APP children in particular, making $\mathbb{A}_c(i) = \emptyset$ by Lemma 3.4.4. By definition of the goal configuration, $\mathbb{A}_c(i) = \mathcal{A}(\tau(\mathbb{G}_c(i)), \tau)$. Combining this with $\mathbb{A}_c(i) = \emptyset$, we deduce that $\tau(\mathbb{G}_c(i)) = \tau$ using the definition of the apply set.

Induction step Let i be a node with APP children $a_1, \ldots, a_n$, attached with the edges $\text{APP}_{\alpha_1}, \ldots, \text{APP}_{\alpha_n}$, respectively. Let i also have MOD children $m_1, \ldots, m_k$, attached with the edges $\text{MOD}_{\beta_1}, \ldots, \text{MOD}_{\beta_k}$, respectively. Let $\lambda = \tau(\mathbb{G}_c(i))$ be the lexical type at i, and $\{\tau\} = \mathbb{T}_c(i)$.

By the definition of the apply set, i reaches term type τ from λ if we can show for all APP children:

(i) i has an APP_α child if and only if $\alpha \in \mathcal{A}(\lambda, \tau)$

(ii) if a is an APP_α child of i, then it has the term type $req_\alpha(\lambda)$.

(i) follows from the goal condition $\mathbb{A}_c(i) = \mathcal{A}(\lambda, \tau)$ and Lemma 3.4.4.
(ii) the only way the edge $i \xrightarrow{\text{APP}_\alpha} a$ can be created is by the $\text{APPLY}(\alpha, a)$ transitions with i on top of the stack. Both transition systems enforce $\mathbb{T}_c(a) = \{req_\alpha(\lambda)\}$. Using the inductive hypothesis on a, it follows that a evaluates to a graph of type $req_\alpha(\lambda)$.

Although the MOD children of i cannot alter the term type of i, they could make the subtree rooted in i ill-typed. That is, for any MOD_β child m that evaluates to a graph of type τ' by the inductive hypothesis, we have to show that $\tau' - \beta \subseteq \lambda \wedge req_\beta(\tau') = [\,]$. The MOD_β edge was created by a $\text{MODIFY}(\beta, m)$ transition. The $\text{MODIFY}(\beta, m)$ transition (in case of LTF) or the next FINISH transition (in case of LTL) resulted in a configuration c', where the term types of m were restricted in exactly that way: $\mathbb{T}_{c'}(m) = \{\tau \in \Omega | \tau - \beta \subseteq \lambda \wedge req_\alpha(\tau) = [\,]\}$. In the derivation from c' to c, a CHOOSE (LTF) or FINISH (LTL) transition must have been applied when m was on top of the stack (because the MOD_β edge was created and c is a goal configuration), which resulted in $\mathbb{T}_c(m) = \{\tau'\}$, where $\tau' \in \mathbb{T}_{c'}(m) = \{\tau \in \Omega | \tau - \beta \subseteq \lambda \wedge req_\alpha(\tau) = [\,]\}$. This means that the well-typedness condition indeed also holds for τ'. □

3.4.2 Completeness

Theorem 3.4.6 (Completeness). For every well-typed AM dependency tree t, there are valid sequences of LTF and LTL transitions that build exactly t.

We do not need any additional assumptions to prove this theorem. The proof is constructive: for any well-typed AM dependency tree t, Algorithms 2 and 3 give transition sequences that, when prefixed with an appropriate INIT operation, generate t. We show this by showing the following lemma (for LTF):

Lemma 3.4.7. Let t be a well-typed AM dependency tree with term type τ whose root is r and let c be a configuration derived by LTF with

(i) $\tau \in \mathbb{T}_c(r)$,

(ii) r is on top of $\mathbb{S}_c$,

(iii) $W_c - O_c \geq |t| - 1$, i.e. $W_c - O_c$ is at least the number of nodes in t without the root,

(iv) $i \notin Dom(\mathbb{G}_c)$ for all nodes i of t, and

(v) $i \notin Dom(\mathbb{E}_c)$ for all nodes $i \neq r$ of t

Then $H_{LTF}(c, t)$ (Algorithm 2) constructs, with valid LTF transitions, a configuration c' such that

(a) c' contains the edges of t,

(b) $\mathbb{G}_{c'}(i) = G_i$ where G_i is the constant at i in t,

(c) $\mathbb{S}_{c'}$ is the same as $\mathbb{S}_c$ but without r on top, i.e. $\mathbb{S}_c = \mathbb{S}_{c'}|r$,

(d) $W_{c'} = W_c - (|t| - 1)$, and

(e) for all j that are *not* nodes of t, none of $\mathbb{A}, \mathbb{G}, \mathbb{T}, \mathbb{E}$ changes, e.g. $\mathbb{A}_{c'}(j) = \mathbb{A}_c(j)$.

The lemma basically says that we can insert t as a subtree into a configuration c with LTF transitions. The conditions (i) and (ii) say that we have already put the root of t on top of the stack and thus can now start to add the rest of t. Condition (iii) says that there are enough words left in the sentence to fit t into c, where -1 comes from the fact that the root of t is already on the stack and has an incoming edge. Conditions (iv) and (v) ensure that the part is still empty where we want to put the subtree.

Theorem 3.4.6 for LTF then follows from applying the lemma to the whole tree t and the configuration obtained after INIT(t). This yields a configuration with empty stack, which is a goal configuration (see Lemma 3.4.1).

Before we approach the proof of Lemma 3.4.7, we need to show the following:

Lemma 3.4.8. Let c be a configuration derived by LTF. If for any token i, $i \notin \mathcal{D}(\mathbb{G}_c)$ then $i \notin \mathcal{D}(\mathbb{A}_c)$.

Proof. We show its contraposition: If for any token i, $i \in \mathcal{D}(\mathbb{A}_c)$ then $i \in \mathcal{D}(\mathbb{G}_c)$. The CHOOSE transition defines $\mathbb{A}_c$ for i, and defines $\mathbb{G}_c$ for i at the same time. There is no transition that can remove i from $\mathcal{D}(\mathbb{G}_c)$. □

Proof of Lemma 3.4.7. By structural induction over t.

Base case Let i be on top of the stack in $\mathbb{S}_c$. t is a leaf with graph constant G, thus $W_c - O_c \geq |t| - 1 = 0$. H_{LTF} returns the sequence $\textsc{Choose}(\tau(G), G), \textsc{Pop}$. It is easily seen that this sequence, if valid, yields a configuration c' where $\mathbb{T}_{c'}(i) = \{\tau(G)\}$, $\mathbb{G}_{c'}(i) = G$ and $\mathbb{A}_{c'}(i) = \mathcal{A}(\tau(G), \tau(G)) = \emptyset$. c' also contains all edges of t (there are none).

In order for $\textsc{Choose}(\tau(G), G)$ to be applicable, it must hold that $\tau(G) \in \mathbb{T}_c(i)$ (holds by (i)), $i \notin \mathcal{D}(\mathbb{G}_c)$ (holds by (iv)) and that $\tau(G) \in PossL(\tau(G), \emptyset, W_c - O_c)$, which is equivalent to

$$|\mathcal{A}(\tau(G), \tau(G))| \leq W_c - O_c$$

Since $\mathcal{A}(\tau(G), \tau(G)) = \emptyset$ and $W_c - O_c \geq 0$, this holds with equality. The transition $\textsc{Choose}(\tau(G), G)$ yields a configuration c_1, where $\mathbb{A}_{c_1}(i) = \mathcal{A}(\tau(G), \tau(G)) = \emptyset$, so we can perform $\textsc{Pop}$, which gives us the configuration c'. Since we have not drawn any edge $W_{c'} = W_c = W_c - (1 - 1) = W_c - (|t| - 1)$. Note that these transitions have not changed any $\mathbb{A}, \mathbb{G}, \mathbb{T}, \mathbb{E}$ for $j \neq i$.

Induction step Let i be on top of the stack in $\mathbb{S}_c$ and let i in t have $\textsc{App}$ children $a_1, \ldots, a_n$, attached with the edges $\textsc{App}_{\alpha_1}, \ldots, \textsc{App}_{\alpha_n}$, respectively, where n might be 0. Let i in t also have $\textsc{mod}$ children $m_1, \ldots, m_k$, attached with the edges $\textsc{mod}_{\beta_1}, \ldots, \textsc{mod}_{\beta_k}$, respectively, where k might be 0 as well. Let G be the constant of i in t, and τ be its term type. By well-typedness of t and the definition of the apply set, we have $\mathcal{A}(\tau(G), \tau) = \{\alpha_1, \ldots, \alpha_n\}$.

$H_{LTF}(t, c)$ returns the sequence

$$c \xrightarrow{\textsc{Choose}(\tau,\, \text{G})} c'_0 \xrightarrow{\textsc{apply}(\alpha_1, a_1)} c_1 \xrightarrow{H_{LTF}(a_1, c_1)} c'_1 \ldots c'_{n-1} \xrightarrow{\textsc{apply}(\alpha_n, a_n)} c_n \xrightarrow{H_{LTF}(a_n, c_n)} c_{n'}$$
$$c'_n \xrightarrow{\textsc{modify}(\beta_1, m_1)} c_{n+1} \xrightarrow{H_{LTF}(m_1, c_{n+1})} c'_{n+1} \ldots \xrightarrow{H_{LTF}(m_k, c_{n+k})} c'_{n+k} \xrightarrow{\textsc{Pop}} c' \tag{3.1}$$

where c_1 is the configuration after $\textsc{Choose}(\tau, G)$, $\textsc{apply}(\alpha_1, a_1)$ etc. For now, let us assume that conditions (i)-(v) are fulfilled for $a_1, \ldots, a_n, m_1, \ldots, m_k$ and their respective configurations and that the sequence is valid. We will verify this at a later stage.

We can apply the inductive hypothesis for all children, which means that c' contains the edges present in the subtrees $a_1, \ldots, a_n, m_1, \ldots, m_k$ and for all nodes j such that j is a descendant of one of $a_1, \ldots, a_n, m_1, \ldots, m_k$, it holds that $\mathbb{G}_{c'}(j) = G_j$ because H_{LTF} applied to some child of t will do the assignment and such an assignment can never be changed in LTF. Assuming the above transition sequence is valid, it is obvious that it also adds the edges from i to $a_1, \ldots, a_n, m_1, \ldots, m_k$ with the correct labels (consequence (a)) and also makes the assignment $\mathbb{G}_{c'}(i) = G_i$ using $\textsc{Choose}(\tau, G)$ (consequence (b)).

Now we go over the transition sequence in Eq. 3.1 and check that the transitions can be applied, the conditions (i)-(v) hold and what happens to the stack.

First, in order for $\textsc{Choose}(\tau(G), \tau)$ to be applicable, it must hold that $\tau \in \mathbb{T}_c(i)$ (holds by (i)), $i \notin \mathcal{D}(\mathbb{G}_c)$ (holds by (iv)) and that $\tau(G) \in PossL(\tau, \emptyset, W_c - O_c)$, which is equivalent to

$$|\mathcal{A}(\tau(G), \tau)| \leq W_c - O_c$$

Since $\mathcal{A}(\tau(G), \tau) = \{\alpha_1, \ldots, \alpha_n\}$ and $W_c - O_c \geq |t| - 1 \geq |\{\alpha_1, \ldots, \alpha_n\}| = n$, this holds. This yields a configuration c'_0 where $\mathbb{T}_{c'_0}(i) = \{\tau\}$ and $\mathbb{A}_{c'_0}(i) = \emptyset$.

Next, we use the transition $\textsc{apply}(\alpha_1, a_1)$. This is allowed because $\alpha_1 \in \mathcal{A}(\tau(G), \tau)$ (see above), $\alpha_1 \notin \mathbb{A}_{c'_0}(i)$ and $a_1 \notin \mathcal{D}(\mathbb{E}_{c'_0})$ (condition (v)). We get a new configuration c_1 where $\mathbb{A}_{c_1}(i) = \{\alpha_1\}$, $\mathbb{T}_{c_1}(a_1) = \{req_{\alpha_1}(\tau(G))\}$ and $\mathbb{S}_{c_1} = \mathbb{S}_{c_0}|a_1$. We now justify why the inductive hypothesis can be used for a_1 and c_1:

By well-typedness of t, we know that $\mathbb{T}_{c_1}(a_1) = \{req_{\alpha_1}(\tau(G))\} = \{\tau_{a_1}\}$ where τ_{a_1} is the term type of a_1 (condition (i)). From the step before, a_1 is on top of the stack in $\mathbb{S}_{c_1}$ (condition (ii)). We use the fact that $j \notin Dom(\mathbb{G}_c)$ for all nodes j of t and $j \notin Dom(\mathbb{E}_c)$ for all nodes $j \neq i$ (our conditions (iv) and (v)) to justify that conditions (iv) and (v) are also met for a_1. What is left to verify is that $W_{c_1} - O_{c_1} \geq |a_1| - 1$. First, note that $W_{c_1} = W_c - 1$ because of the App_{α_1} edge. We can decompose O_{c_1} as follows:

$$O_{c_1} = O_c - O_c(i) + O_{c_1}(i)$$

because we have only changed $\mathbb{G}_c$ and $\mathbb{A}_c$ for i, not for any other token. $O_c(i) = 0$ by Lemma 3.4.8 and $i \notin \mathcal{D}(\mathbb{G}_c)$ (condition (iv)). We can also see that $O_{c_1}(i) = n - 1$ by definition of $O(\cdot)$ and taking into account that we have drawn the App_{α_1} edge and thus $\mathbb{A}_{c_1} = \{\alpha_1\}$. This means that

$$W_{c_1} - O_{c_1} = (W_c - 1) - (O_c + n - 1) = W_c - O_c - n$$

From condition (iii), we know that $W_c - O_c \geq |t| - 1$. Since t consists of node i and at least n children a_j each of which has $|a_j| - 1$ nodes, we have that

$$|t| \geq 1 + n + \sum_{j=1}^{n}(|a_j| - 1)$$

which is equivalent to

$$|t| - 1 \geq n + \sum_{j=1}^{n}(|a_j| - 1) \tag{3.2}$$

Plugging this together, we get

$$W_{c_1} - O_{c_1} = W_c - O_c - n \geq \sum_{j=1}^{n}(|a_j| - 1) \geq |a_1| - 1$$

After $H_{LTF}(a_1, c_1)$ we get a configuration c_1'. We have just argued that the inductive hypothesis applies for $H_{LTF}(a_1, c_1)$, so we can use it and find that we are in a nearly identical situation as before $\text{APPLY}(\alpha_1, a_1)$: The stack is $\mathbb{S}_{c_1'} = \mathbb{S}_{c_1}|a_1 = \mathbb{S}_c$. That is, in $\mathbb{S}_{c_1'}$ the top of the stack is i again. What has changed is $W_{c_1'} - O_{c_1'}$ and of course $\mathbb{A}_{c_1'} = \{\alpha_1\}$, which was empty before. We can now apply $\text{APPLY}(\alpha_2, a_2)$ and continue.

Let us consider the general case for $H_{LTF}(a_l, c_l)$ with $1 \leq l \leq n$ where we are in c_l arriving from $\text{APPLY}(\alpha_l, a_l)$. At this point, we know

(i) $\mathbb{T}_{c_l} = \{\tau_{a_l}\}$ where τ_{a_l} is the term type of a_l (by APPLY before)

(ii) i is on top of the stack (inductive hypothesis for $l' < l$)

In effect, conditions (i) and (ii) for the inductive hypothesis for $H_{LTF}(a_l, c_l)$ are met. Conditions (iv) and (v) for a_l are fulfilled by our assumptions (iv) and (v) because a_l is a subtree of i. What remains to be checked is $W_{c_l} - O_{c_l} \geq |a_l| - 1$. We can calculate $W_{c_l} = W_c - l - \sum_{j=1}^{l-1}(|a_j| - 1)$, where the summation over j comes from the inductive hypothesis for the children $j < l$ and $-l$ comes from the APPLY transitions we have performed. O_{c_l} is simply $O_{c_l} = O_c + n - l$ because the CHOOSE transition resulted in

$O_{c'_0} = O_c + n$ and we have drawn l App edges already. Plugging this together, we get

$$\begin{aligned}
W_{c_l} - O_{c_l} &= W_c - l - \sum_{j=1}^{l-1}(|a_j| - 1) - (O_c + n - l) \\
&\geq (|t| - 1) - n - \sum_{j=1}^{l-1}(|a_j| - 1) \\
&\geq \sum_{j=l}^{n}(|a_j| - 1) \geq |a_l| - 1
\end{aligned}$$

where the first step replaces $W_c - O_c$ by $|t| - 1$ (assumption (iii)) and the second step replaces $(|t| - 1)$ using Eq. 3.2.

A similar line of reasoning can be used to justify the use of the inductive hypothesis for $H_{LTF}(m_1, c_{n+1}), \ldots, H_{LTF}(m_1, c_{n+k})$.

Note that by applying the inductive hypothesis to all children, we know that i is always on top of the stack after H_{LTF} was applied. This justifies the final Pop transition, because at that point $\mathbb{A}_{c'_{n+k}} = \mathcal{A}(\tau(G), \tau)$. Consequence (c) follows from this Pop.

We did not change any of $\mathbb{E}, \mathbb{A}, \mathbb{T}, \mathbb{G}$ outside of our subtree i (consequence (e)). This follows from the inductive hypotheses of the children and the fact that i was always on top of the stack when we performed any transition.

If we want to determine $W_{c'}$, we note that we have drawn $n + k$ edges and for each child $ch \in a_1, \ldots, a_n, m_1, \ldots m_k$, we know by the inductive hypothesis that this has drawn $|ch| - 1$ edges. In total, we have

$$\begin{aligned}
W_{c'} &= W_c - \left[\sum_{j=1}^{n}(|a_j| - 1) + \sum_{j=1}^{k}(|m_j| - 1)\right] - (n + k) \\
&= W_c - \left[\sum_{j=1}^{n}|a_j| + \sum_{j=1}^{k}|m_j| - (n + k)\right] - (n + k) \\
&= W_c - (|t| - 1)
\end{aligned}$$

where the last step makes use of the fact that $|t| = 1 + \sum_{j=1}^{n}|a_j| + \sum_{j=1}^{k}|m_j|$.

□

For LTL, the same principle applies with a near identical lemma which only also asks that for the root r of t, $\mathbb{A}_c(r) = \emptyset$. The procedure to construct the transition sequence is shown in Algorithm 3.

3.4.3 No dead ends

For both LTF and LTL, the following theorem guarantees that we can always get a complete analysis for a sentence:

Theorem 3.4.9 (No dead ends). If c is a configuration derived by LTF or LTL then there is a valid sequence of transitions that brings c to a goal configuration c'.

Together with the soundness theorem (Theorem 3.4.3) that every goal configuration corresponds to well-typed AM dependency tree, this means that we can always finish a derivation to get a well-typed AM dependency tree, no matter what the sentence is or

Algorithm 2 Generate LTF transitions for AM dependency tree

```
 1: function $H_{LTF}(c, t)$
 2:     Let $t$ have graph constant $G$
 3:     and term type $\tau$
 4:     $c \leftarrow \textsc{Choose}(\tau, G)(c)$
 5:     for $\textsc{App}_\alpha$ child $a$ of t do
 6:         $c \leftarrow \textsc{apply}(\alpha, a)(c)$
 7:         $c \leftarrow H_{LTF}(c, a)$
 8:     end for
 9:     for $\textsc{mod}_\beta$ child $m$ of $t$ do
10:         $c \leftarrow \textsc{modify}(\beta, m)(c)$
11:         $c \leftarrow H_{LTF}(c, m)$
12:     end for
13:     $c \leftarrow \textsc{Pop}(c)$
14:     return $c$
15: end function
```

Algorithm 3 Generate LTL transitions for AM dependency tree

```
 1: function $H_{LTL}(c, t)$
 2:     Let $t$ have graph constant $G$
 3:     for $\textsc{App}_\alpha$ child $a$ of t do
 4:         $c \leftarrow \textsc{apply}(\alpha, a)(c)$
 5:     end for
 6:     for $\textsc{mod}_\beta$ child $m$ of $t$ do
 7:         $c \leftarrow \textsc{modify}(\beta, m)(c)$
 8:     end for
 9:     $c \leftarrow \textsc{Finish}(G)(c)$
10:     Let $t_1, \ldots t_n$ be the children of $t$
11:     on the stack in $c$
12:     for $i \in 1, \ldots, n$ do
13:         $c \leftarrow H_{LTL}(c, t_i)$
14:     end for
15:     return $c$
16: end function
```

how the transitions are scored. The proof of Theorem 3.4.9 is constructive both for LTF and LTL and is given below. In both cases, we proof a lemma first that there are always "enough" words left.

Theorem 3.4.9 only holds if we make a few assumptions that are mild in practice. Recall that we assumed that we are given a set of graph constants C that can draw source names from a set $\mathcal{S}$, a set of types Ω and a set of edge labels Lab. We now make the following assumptions about their relationships:

Assumption 1. For all types $\lambda \in \Omega$, there is a constant $G \in C$ with type $\tau(G) = \lambda$.

Assumption 2. For all types $\lambda \in \Omega$ and all source names $\alpha \in \mathcal{S}$, if $req_\alpha(\lambda)$ is defined then $req_\alpha(\lambda) \in \Omega$.

Assumption 3. If $\textsc{mod}_\alpha \in Lab$ then $[\alpha] \in \Omega$.

Assumption 4. For all source names $\alpha \in \mathcal{S}$, $\textsc{App}_\alpha \in Lab$.

Assumption 5. There are *no* constraints imposed on which graph constants can be assigned to a particular word.

While the last assumption deviates from traditional grammar engineering, it is not unique to this work. In particular, no such constraints are made by existing AM dependency parsers and the assumption is often made in TAG or CCG parsing when word embeddings are used in predicting supertags to generalize to words unseen during training (Lewis and Steedman, 2014; Kasai et al., 2017).

The assumptions made are almost perfectly met in practice, requiring minimal action to make them hold (see Section 5.2).

In the proof of Theorem 3.4.9 we want to use the fact $[\,] \in \Omega$; this follows from the assumptions above:

Lemma 3.4.10. The empty type $[\,] \in \Omega$.

Proof. Assumption 2 says that for all types $\lambda \in \Omega$ and all sources $\alpha \in \mathcal{S}$, the type $req_\alpha(\lambda)$ (if defined) is also a member of Ω. Since types are formally DAGs, each type τ is either empty (that is: $[\,]$) or has a node n without outgoing edges. In the latter case, $req_n(\tau) = [\,]$. □

LTF

We prove a lemma that there are always at most as many sources that we have still to fill as there are words without incoming edges.

Lemma 3.4.11. For all configurations derived with LTF, $O_c \leq W_c$.

Proof. By structural induction over the derivation.

Base case The initial state c does not define $\mathbb{A}$ for any token, thus $O_c(i) = 0$ for all i. The number of words without incoming edges in configuration c is $W_c \geq 1$. Therefore, $\sum_i O_c(i) = O_c \leq W_c$.

Induction step Inductive hypothesis: $O_c \leq W_c$
Goal: $O_{c'} \leq W_{c'}$ where c' derives in one step from c.
The derivation step from c to c' is one of:

Init(i) After init, $\mathbb{A}_{c'}$ is not defined for any i, thus $O_c = 0$.

Pop This transition only changes the stack, which does not affect O, so $O_{c'}(i) = O_c(i)$ for all i and $W_{c'} = W_c$. The inductive hypothesis applies.

Choose(τ, G) Let i be the active node. No edge was created, thus $W_{c'} = W_c$. For all $j \neq i$, $O_{c'}(j) = O_c(j)$. We can thus write $O_{c'}$ as

$$O_{c'} = O_c - O_c(i) + O_{c'}(i)$$

Since Choose(τ, G) was applicable in c, we know that $i \notin \mathcal{D}(\mathbb{G}_c)$. By Lemma 3.4.8 and by definition of $PossL$, we have that $O_c(i) = 0$, so

$$O_{c'} = O_c + O_{c'}(i) \tag{3.3}$$

We now look into the value of $O_{c'}(i)$. Since CHOOSE was applied, we know that $\mathbb{G}_{c'}(i) = G$, $\mathbb{A}_{c'}(i) = \emptyset$ and that $\tau(G) \in PossL(\tau, \emptyset, W_c - O_c)$, which simplifies to $|\mathcal{A}(\tau(G), \tau)| \leq W_c - O_c$. From this follows that $O_{c'}(i) = \min_{\lambda' \in \{\tau(G)\}, \tau' \in \mathbb{T}_{c'}(i)} |\mathcal{A}(\lambda', \tau') - \mathbb{A}_{c'}(i)| \leq W_c - O_c$. Substituting this for $O_{c'}(i)$ in Eq. 3.3, we get

$$\begin{aligned} O_{c'} &= O_c + O_{c'}(i) \\ &\leq O_c + W_c - O_c = W_c = W_{c'} \end{aligned}$$

APPLY(α, j) Let i be the active node. Since an edge to j was created in the transition, $W_{c'} + 1 = W_c$. We decompose $O_{c'}$ again:

$$O_{c'} = O_c - O_c(i) + O_{c'}(i)$$

Since APPLY could be performed, we know that $\mathbb{T}_c$ and $\mathbb{G}_c$ are defined for i and let us denote them $\mathbb{T}_c(i) = \{\tau\}$ and $\mathbb{G}_c(i) = G$. Thus, $O_c(i) = |\mathcal{A}(\tau(G), \tau) - \mathbb{A}_c(i)|$. Since the precondition of APPLY said that $\alpha \notin \mathbb{A}_c(i)$ and APPLY has the effect that $\mathbb{A}_{c'}(i) = \mathbb{A}_c(i) \cup \{\alpha\}$, we know that $O_{c'}(i) = |\mathcal{A}(\tau(G), \tau) - (\mathbb{A}_c(i) \cup \{\alpha)\}| < O_c(i)$. This means that $O_{c'} < O_c$. Using the inductive hypothesis $O_c \leq W_c$ and $W_{c'}+1 = W_c$, we get

$$O_{c'} < O_c \leq W_{c'} + 1$$

which means that $O_{c'} \leq W_{c'}$.

MODIFY(β, j) Let i be the active node. In Section 3.2.2, we made the restriction that MODIFY is only applicable if

$$W_c - O_c \geq 1 \tag{3.4}$$

The transition created an edge, which means that $W_{c'} = W_c - 1$. $O_{c'}$ depends on $\mathbb{G}_{c'}, \mathbb{A}_{c'}$ and $\mathbb{T}_{c'}$. The only thing that changed from c to c' is that $\mathbb{T}_{c'}$ is now defined for j. However, $\mathbb{A}_{c'}$ is still not defined for j, so $O_{c'}(j) = O_c(j) = 0$. This means $O_{c'} = O_c$. Substituting those into Eq. 3.4 and re-arranging, we get $O_{c'} \leq W_{c'}$.

□

We now show that there are no dead ends by showing that for any configuration c derived by LTF, we can construct a valid sequence of transitions such that the stack becomes empty. By Lemma 3.4.1 this means that c is a goal configuration. We empty the stack by repeatedly applying Algorithm 4.

In line 17, we compute the sources that we still have to fill in order to pop i off the stack. We assume an arbitrary order and o_j refers to one particular source in o. The symbol $\oplus$ denotes concatenation.

Lemma 3.4.12. For any configuration c, $C_{LTF}(c)$ (Algorithm 4) generates a valid sequence s of LTF transitions such that ($|\mathbb{S}_{c'}| < |\mathbb{S}_c|$ or $|\mathbb{S}_{c'}| = 0$) and there is a token i for which $\mathbb{G}_{c'}(i)$ is defined, where c' is the configuration obtained by applying s to c.

Proof. First, we show that $\mathbb{G}_{c'}$ is defined for some i in c'. We make a case distinction based on in which line the algorithm returns. If it returns in lines 4, 11 or 14, it is obvious that $\mathbb{G}_{c'}$ is defined for some i. If it returns in line 26 then o is non-empty because $O_c(i) > 0$. If o is non-empty, we use a CHOOSE transition in the for-loop. The remaining case is returning in line 6. Note that the stack is empty but it is not the initial configuration (otherwise, we would have returned in line 4), so an INIT transition must have been applied, which pushes a token to the stack. Since the stack is now empty in c', a POP transition must have been applied, which is only applicable if $\mathbb{G}$ is defined for the item on top of the stack. Consequently, $\mathbb{G}_{c'}$ is defined for some i.

Further, note that every path through Algorithm 4 either reduces the size of the stack (one more POP transition than tokens pushed to the stack by APPLY) or keeps it effectively empty.

C_{LTF} is constructed in a way that the transition sequence is valid. However, there are a few critical points:

- In line 3, we assume the existence of a graph constant $G \in C$ with $\tau(G) = [\,]$. This follows from Lemma 3.4.10 and Assumption 1.
- In line 13, it is assumed that there exists a graph constant $G \in C$ with $\tau(G) \in \mathbb{T}_c(i)$. This graph constant always exists because either $\mathbb{T}_c(i)$ is a request (if i has an incoming APP edge) and thus by Assumptions 1 and 4 there is a graph constant $G \in C$, or $\mathbb{T}_c(i)$ is a set of types resulting from a MODIFY transition. Here, the existence of a suitable graph constant G with type $\tau(G) \in \mathbb{T}_c(i)$ follows from Assumptions 1 and 3. Assumption 5 makes explicit that there are no further constraints on how we choose G.
- In line 20, it is assumed that there exist $|o|$ tokens without incoming edges. This is true because $|o| = O_c(i) \leq \sum_j O_c(j) = O_c$ and by Lemma 3.4.11, it follows that $|o| \leq W_c$, showing that there are indeed enough tokens without incoming edges.
- In line 24, it is assumed that $\text{APP}_{a_i} \in \mathit{Lab}$ for some source o_j; this is guaranteed by Assumption 4.

□

In summary, we can turn any configuration c derived by LTF into a goal configuration by repeatedly applying C_{LTF} to it until the (finite) stack is empty. By Lemma 3.4.1, this is a goal configuration.

LTL

The proof works similarly. We first prove a similar lemma that if i is the active node, $O_c(i) \leq W_c$ and then construct a function C_{LTL} (see Algorithm 5) that produces a valid sequence of transitions that we repeatedly apply to reach a goal configuration.

Lemma 3.4.13. Let c be a configuration derived by LTL. If i is the active node in c, then $O_c(i) \leq W_c$.

Proof. By structural induction over the derivation.

Base case In the initial state, the stack $\mathbb{S}_c$ is empty, making the antecedent of the implication false for all i and thus the implication true.

Algorithm 4 Complete LTF sequence

```
1:  function $C_{LTF}(c)$
2:      if $c = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ then
3:          Let $G \in C$ with $\tau(G) = [\,]$.
4:          return $\textsc{Init}(1), \textsc{choose}([\,], G), \textsc{Pop}$
5:      end if
6:      if $\mathbb{S}_c = []$ then return $[]$
7:      end if
8:      Let $i$ be top of $\mathbb{S}_c$.
9:      if $O_c(i) = 0$ then
10:         if $i \in Dom(\mathbb{G}_c)$ then
11:             return $\textsc{Pop}$
12:         else
13:             Let $G \in C, \tau(G) \in \mathbb{T}_c(i)$.
14:             return $\textsc{choose}(\tau(G), G), \textsc{Pop}$
15:         end if
16:     end if
17:     Let $o = \mathcal{A}(\mathbb{G}_c(i), \tau) - \mathbb{A}_c(i)$
18:     where $\mathbb{T}_c(i) = \{\tau\}$
19:     Let $\rho_j = req_{o_j}(\tau(\mathbb{G}_c(i)))$
20:     Let $a_1, \ldots, a_{|o|}$ be tokens without heads
21:     $s = []$
22:     for $a_j \in a_1, \ldots, a_{|o|}$ do
23:         Let $G$ be a constant of type $\rho_j$
24:         $s = s \oplus \textsc{apply}(o_j, a_j), \textsc{Choose}(\rho_j, G), \textsc{Pop}$
25:     end for
26:     return $s \oplus \textsc{Pop}$
27: end function
```

Induction step Inductive Hypothesis: If i is the active node in c, then $O_c(i) \leq W_c$. Goal: If i is the active node in c', then $O_{c'}(i) \leq W_{c'}$ where c' derives in one step from c. The applied transition is one of:

$\textsc{Init}(i)$ The previous configuration c must be the initial configuration. Now i is the active node in c' and $\mathbb{A}_{c'}(i) = \emptyset$ and $\mathbb{T}_{c'}(i) = \{[\,]\}$ and $\mathbb{G}_{c'}$ is not defined for i. Then $O_{c'}(i) = \min_{\lambda \in \Omega} |\mathcal{A}(\lambda, [\,]) - \mathbb{A}_{c'}(i)|$. Note that the empty type $[\,] \in \Omega$ by lemma 3.4.10 and that $\mathcal{A}([\,], [\,]) = \emptyset$. Choosing $\lambda = [\,]$, we get $O_{c'}(i) = 0$. $\textsc{Init}(i)$ created an edge into i, so $W_{c'} = W_c - 1$. Since a sentence consists of at least one word ($W_c \geq 1$), we have $O_{c'}(i) = 0 \leq W_{c'}$.

$\textsc{Apply}(\alpha, j)$ Let i be the active node in c. Then, by construction of $\textsc{Apply}(\alpha, j)$ it remains the active node in c'. After the transition, $\mathbb{T}_{c'}(i) = \mathbb{T}_c(i)$, $\mathbb{A}_{c'}(i) = \mathbb{A}_c(i) \cup \{\alpha\}$. Thus, $O_{c'}(i)$ can be written as follows:

$$O_{c'}(i) = \min_{\lambda' \in \Omega, \tau' \in \mathbb{T}_c(i)} |\mathcal{A}(\lambda', \tau') - (\mathbb{A}_c(i) \cup \{\alpha\})|$$

Since $\textsc{Apply}(\alpha, j)$ was applicable, the pre-conditions must be fulfilled, i.e.

$$\exists \lambda \in \Omega. \exists \tau \in \mathbb{T}_c(i). \lambda \in PossL(\tau, \mathbb{A}_c(i) \cup \{\alpha\}, W_c - 1)$$

Expanding the definition of $PossL$ we get:

$$\mathbb{A}_c(i) \cup \{\alpha\} \subseteq \mathcal{A}(\lambda, \tau) \wedge |\mathcal{A}(\lambda, \tau) - (\mathbb{A}_c(i) \cup \{\alpha\})| \leq W_c - 1$$

for some $\lambda \in \Omega$ and $\tau \in \mathbb{T}_c(i)$. If we now choose $\lambda' = \lambda$ and $\tau' = \tau$ in $O_{c'}(i)$, we get

$$O_{c'}(i) \leq |\mathcal{A}(\lambda, \tau) - (\mathbb{A}_c(i) \cup \{\alpha\})| \leq W_c - 1$$

Since $W_{c'} = W_c - 1$, it holds that $O_{c'}(i) \leq W_{c'}$.

MODIFY(β, j) Let i be the active node. It also remains the active node in c'. The transition consumes a word, that is $W_{c'} = W_c - 1$. However, it can only be applied if $W_c - O_c \geq 1$. Since O_c is obtained by summing over all tokens, $O_c(i) \leq O_c$. We get:

$$O_c(i) \leq O_c \leq W_c - 1 = W_{c'}.$$

Finally, $O_{c'}(i) = O_c(i)$ because none of $\mathbb{A}, \mathbb{G}, \mathbb{T}$ changed for i during the MODIFY(β, j) transition.

FINISH(G) Let i be active node *after* the transition, that is, in c'. The FINISH transition presupposes that i has an incoming edge. We distinguish two cases based on the label:

- i has an incoming APP$_\alpha$ edge. Then we have that $\mathbb{T}_{c'}(i) = \{req_\alpha(\tau(G))\}$ and $\mathbb{G}_{c'}$ undefined for i. Then $O_{c'}(i) = \min_{\lambda \in \Omega} |\mathcal{A}(\lambda, req_\alpha(\tau(G)))|$. By Assumption 2, $req_\alpha(\tau(G)) \in \Omega$ and by definition of the apply set $\mathcal{A}(\lambda, \lambda) = \emptyset$ for all types λ, so in particular also for $req_\alpha(\tau(G))$, which makes $O_{c'}(i) = 0$.
- i has an incoming MOD$_\beta$ edge. By Assumption 3, we know that $[\beta] \in \Omega$, for which $[\beta] \in \mathbb{T}_{c'}(i)$ holds by construction of FINISH(G). Expanding the definition of $O_{c'}(i)$, we get: $O_{c'}(i) = \min_{\lambda \in \Omega, \tau' \in \mathbb{T}_{c'}(i)} |\mathcal{A}(\lambda, \tau')|$. By choosing $\lambda = [\beta] = \tau'$, we get $O_{c'}(i) = 0$.

Since $O_{c'}(i) = 0$, it also holds that $O_{c'}(i) \leq W_c = W_{c'}$.

□

Lemma 3.4.14. For a sentence with n words, a valid LTL transition sequence can contain at most n FINISH transitions.

Proof. By contradiction. Assume there is a valid transition sequence s that contains $m > n$ FINISH transitions.
Since FINISH can only be applied when there is some token on the stack and there are more FINISH transitions than there are tokens, FINISH must have been applied twice with the same active node. Since FINISH removes the active node from the stack, i must have been pushed twice. This means that i has two incoming edges. When the second incoming edge was drawn into i the condition $i \notin \mathcal{D}(\mathbb{E})$ was violated, which contradicts the assumption that the transition sequence s is valid.

□

Lemma 3.4.15. Let c be a configuration derived by an LTL transition sequence s that contains j FINISH transitions. Then $C_{LTL}(c)$ (Algorithm 5) generates a valid sequence s' of LTL transitions that leads to a goal configuration c' or $s \oplus s'$ contains $j + 1$ FINISH transitions.

Algorithm 5 Complete LTL sequence

```
1:  function $C_{LTL}(c)$
2:      if $c = \langle\emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ then
3:          Let $G \in C$ with $\tau(G) = [\,]$
4:          return $\textsc{Init}(1), \textsc{Finish}(G)$
5:      end if
6:      if $\mathbb{S}_c = []$ then return $[]$
7:      end if
8:      Let $i$ be top of $\mathbb{S}_c$.
9:      Let $\lambda, \tau$ be the minimizers of $O_c(i) = \min_{\lambda \in \Omega, \tau \in \mathbb{T}_c(i)} |\mathcal{A}(\lambda, \tau) - \mathbb{A}_c(i)|$
10:     if $O_c(i) = 0$ then
11:         Let $G \in C$ with $\tau(G) = \lambda$
12:         return $\textsc{finish}(G)$
13:     end if
14:     Let $o = \mathcal{A}(\lambda, \tau) - \mathbb{A}_c(i)$
15:     Let $\rho_i = req_{o_i}(\tau(\mathbb{G}_c(i)))$
16:     Let $a_1, \ldots, a_{|o|}$ be tokens without heads
17:     $s = []$
18:     for $a_j \in a_1, \ldots, a_{|o|}$ do
19:         $s = s \oplus \textsc{apply}(o_j, a_j)$
20:     end for
21:     $s = s \oplus \textsc{Finish}(G)$ where $\tau(G) = \lambda$
22:     return $s$
23: end function
```

Proof. We first show the main claim and then verify that the generated transition sequence s' is valid. We make a case distinction on the content of the stack in c'.

$\mathbb{S}_{c'}$ is empty We show that c' is a goal configuration. In order to apply Lemma 3.4.2, we have to show that $\mathbb{G}_{c'}$ is defined for some token i. There is only one path through Algorithm 5 that does not assign a graph constant to a token (line 6). Returning in line 6 means that the stack is empty but the state is not the initial state – so something has been removed from the stack already with a FINISH transition. Consequently, $\mathbb{G}$ is defined for some i.

$\mathbb{S}_{c'}$ is not empty Since the stack is not empty, this means the algorithm returns in line 12 or in line 22. Clearly, the transition sequence that the algorithm returns contains a FINISH transition. Together with the j FINISH transitions that have been performed up to the configuration c, this makes $j + 1$ FINISH transitions.

Algorithm 5 is constructed such that it only produces valid transition sequences. However, there are a few critical points:

- Line 3 assumes the existence of a graph constant $G \in C$ with $\tau(G) = [\,]$. This follows from Lemma 3.4.10 and Assumption 1. Assumption 5 explicitly allows us to assign G to any token.

- Line 9 assumes that $K_c(i) = \Omega$ and that $i \in \mathcal{D}(\mathbb{A}_c)$ and $i \in \mathcal{D}(\mathbb{T}_c)$. This is true because i is on top of the stack. $\mathbb{A}$ and $\mathbb{T}$ are always defined for the active node in LTL. $\mathbb{G}$ is never defined for the active node in LTL.

- Lines 11 and 21 assume the existence of a graph constant $G \in C$ of type $\tau(G) = \lambda \in \Omega$, which is guaranteed by Assumption 1. Assumption 5 explicitly allows us to assign G to any token.
- Line 16 assumes that there are at least $|o|$ tokens without incoming edges ($W_c \geq |o|$). This is indeed the case, because $|o| = O_c(i)$ and $O_c(i) \leq W_c$ by Lemma 3.4.13.
- Line 19 assumes that $\textsc{App}_{o_j} \in \mathit{Lab}$. This is guaranteed by Assumption 4.

$\square$

We can construct the transition sequence for which Theorem 3.4.9 asks by repeatedly applying C_{LTL} to a given configuration c. Lemma 3.4.15 shows that applying C_{LTL} to a configuration results either in a goal configuration or increases the number of FINISH transitions by one. Lemma 3.4.14 tells us that there is an upper bound on how many times we can increase the number of FINISH transitions in a valid transition sequence. Since C_{LTL} returns only valid transition sequences, this means that we reach a goal configuration by finitely many applications of C_{LTL}.

Chapter 4

Parsing model

We phrase AM dependency parsing as finding the most likely sequence $d^{(1)}, \dots, d^{(N)}$ of LTF or LTL transitions given an input sentence $\mathbf{x}$:

$$P_\theta(d^{(1)}, \dots, d^{(N)}|\mathbf{x}) = \prod_{t=1}^{N} P_\theta(d^{(t)}|d^{(<t)}, \mathbf{x}) \tag{4.1}$$

In this chapter, we first factorize $P_\theta(d^{(t)}|d^{(<t)}, \mathbf{x})$ in decisions about edges, graph constants and term types (Section 4.1) and then discuss how these can be computed by extending Ma et al.'s model (Section 4.2). We then look at how this model can be trained efficiently (Section 4.3) and how to use a trained model to predict a good transition sequence for a given sentence at test time (Section 4.4).

The models and algorithms described here are implemented in Python, relying on the libraries PyTorch (Paszke et al., 2019) and AllenNLP (Gardner et al., 2017) and are available on github: `https://github.com/coli-saar/topdown-parser`.

4.1 Probability models

Following the model of Ma et al. (2018), we encode the input sentence $\mathbf{x}$ and the entire history of transitions $d^{(<t)}$ into a vector $\mathbf{h}^{(t)}$, conditioned on which we predict the next transition $d^{(t)}$. We will make this dependence explicit here and write $P_\theta(d^{(t)}|\mathbf{h}^{(t)})$.

In practice, there is a small difference between the transitions we use and the ones presented in Chapter 3: we separate the prediction of graph constants into predicting de-lexicalized graph constants and lexical labels (see Section 2.2.2). As a consequence, we have to incorporate the lexical label into the transitions and $\text{C}\textsc{hoose}(\tau, G)$ becomes $\text{C}\textsc{hoose}(\tau, G, lex)$ and $\text{F}\textsc{inish}(G)$ becomes $\text{F}\textsc{inish}(G, lex)$.

4.1.1 LTF

At the first step ($t = 1$), we have to use an Init transition, for which we model the probability as

$$P_\theta(\text{I}\textsc{nit}(i)|\mathbf{h}^{(1)}) = a_i^{(1)}$$

where $a^{(1)}$ is a distribution over the tokens in the sentence. We will describe in Section 4.2 how it is computed. Afterwards, Init is no longer applicable according to the rules of LTF and is assigned probability 0.

Let $\delta^{(t)}$ be 1 if after $d^{(1)}, \dots, d^{(t-1)}$ the Choose transition is allowed and thus required, and 0 otherwise. Then for any edge label $\ell \in Lab$, graph constant $G \in C$, type $\tau \in \Omega$ and lexical label lex, we have the transition probabilities:

$$\begin{aligned}
&P_\theta(\textsc{Apply}(\alpha, j)|\mathbf{h}^{(t)}, \delta^{(t)}) = (1-\delta^{(t)})a_j^{(t)} \cdot P_\theta(\textsc{App}_\alpha|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Modify}(\beta, j)|\mathbf{h}^{(t)}, \delta^{(t)}) = (1-\delta^{(t)})a_j^{(t)} \cdot P_\theta(\textsc{mod}_\beta|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Pop}|\mathbf{h}^{(t)}, \delta^{(t)}) = (1-\delta^{(t)})a_0^{(t)}\\
&P_\theta(\textsc{Choose}(\tau, G, lex)|\mathbf{h}^{(t)}, \delta^{(t)}) = \delta^{(t)} P_\theta(\tau|\mathbf{h}^{(t)}) \cdot P_\theta(G|\mathbf{h}^{(t)}) \cdot P_\theta(lex|\mathbf{h}^{(t)})
\end{aligned}$$

We model Pop as selecting the artificial root token in position 0. This is a difference to Ma et al. (2018) who model a Pop-like transition with selecting the active node.

In practice, we use the shortened version of the transition system (see Section 3.2.3) because it promises faster run times. That is, instead of the Choose(τ, G, lex), we only have transitions that combine Choose immediately with a subsequent transition. The probabilities are basically the same as the product of the Choose(τ, G, lex) transition with the other ones:

$$\begin{aligned}
&P_\theta(\textsc{Choose}(\tau, G, lex),\textsc{Apply}(\alpha, j) \mid \mathbf{h}^{(t)}, \delta^{(t)})\\
&\qquad = \delta^{(t)} P_\theta(\tau|\mathbf{h}^{(t)}) P_\theta(G|\mathbf{h}^{(t)}) P_\theta(lex|\mathbf{h}^{(t)}) \cdot a_j^{(t)} P_\theta(\textsc{App}_\alpha|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Choose}(\tau, G, lex),\textsc{Modify}(\beta, j) \mid \mathbf{h}^{(t)}, \delta^{(t)})\\
&\qquad = \delta^{(t)} P_\theta(\tau|\mathbf{h}^{(t)}) P_\theta(G|\mathbf{h}^{(t)}) P_\theta(lex|\mathbf{h}^{(t)}) \cdot a_j^{(t)} P_\theta(\textsc{mod}_\beta|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Choose}(\tau, G, lex),\textsc{Pop} \mid \mathbf{h}^{(t)}, \delta^{(t)}) = \delta^{(t)} P_\theta(\tau|\mathbf{h}^{(t)}) P_\theta(G|\mathbf{h}^{(t)}) P_\theta(lex|\mathbf{h}^{(t)}) \cdot a_0^{(t)}
\end{aligned}$$

Note that everything conditions on $\mathbf{h}^{(t)}$ as opposed to $\mathbf{h}^{(t)}$ for Choose and $\mathbf{h}^{(t+1)}$ for the next transition. We will see in Section 4.2 that this makes sense because Choose would not provide new information in the step from $\mathbf{h}^{(t)}$ to $\mathbf{h}^{(t+1)}$ anyway. Another difference to the formulation before is that the term $(1-\delta^{(t)})$ is missing since we know that there cannot be a second Choose transition right after the first one and $\delta^{(t)} \cdot (1-\delta^{(t)})$ is always 0.

4.1.2 LTL

Analogously to LTF, we have to use an Init transition at the first step ($t = 1$):

$$P_\theta(\textsc{Init}(i)|\mathbf{h}^{(1)}) = a_i^{(1)}$$

Afterwards, Init is no longer applicable according to the rules of LTL and is assigned probability 0.

For all further transitions with $t > 1$, the probability of transition $d^{(t)}$ given $d^{(<t)}$ is computed as follows:

$$\begin{aligned}
&P_\theta(\textsc{Apply}(\alpha, j)|\mathbf{h}^{(t)}) = a_j^{(t)} \cdot P_\theta(\textsc{App}_\alpha|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Modify}(\beta, j)|\mathbf{h}^{(t)}) = a_j^{(t)} \cdot P_\theta(\textsc{mod}_\beta|\mathbf{h}^{(t)}, tos \to j)\\
&P_\theta(\textsc{Finish}(G, lex)|\mathbf{h}^{(t)}) = a_0^{(t)} \cdot P_\theta(G|\mathbf{h}^{(t)}) \cdot P_\theta(lex|\mathbf{h}^{(t)})
\end{aligned}$$

Analogously to LTF, we model Finish as selecting the artificial root token in position 0 and selecting a graph constant $G \in C$. Note that we do not need to predict term types for

LTL because no transition is parameterized by a term type.

4.2 Architecture

The architecture of the parser is similar to that of Ma et al. (2018) but with some differences that take particular aspects of the semantic nature of the parsing task into account. The model consists also of a bidirectional LSTM encoder and a LSTM decoder. At every step of the decoder, we use an attention mechanism to point to a certain token in the input sentence and use the attention score to model transition probabilities. An important difference to Ma et al. (2018) is that we predict graph constants, lexical types, and – for LTF – term types at every step as well. We now go over the architecture in detail:

We represent a token x_i as follows:

$$\mathbf{x}'_i = [E(i, \mathbf{x}), E(p_i), E(l_i), E(\mathrm{ne}_i), \mathrm{CharCNN}(x_i)]$$

where p_i, l_i and ne_i are POS tag, lemma and named entity tag, respectively, which E maps to embeddings. $E(i, \mathbf{x})$ are either GloVe embeddings (which simply embed $\mathbf{x}_i$) or a learned weighted average over the different layers of BERT.

We encode the sentence with two different multi-layer biLSTMs yielding sequences of hidden states $\mathbf{s}_1, \ldots \mathbf{s}_n$ and $\mathbf{s'}_1, \ldots \mathbf{s'}_n$. The states $\mathbf{s}_1, \ldots \mathbf{s}_n$ will be used to predict edges analogously to Ma et al. (2018), while $\mathbf{s'}_1, \ldots \mathbf{s'}_n$ are used to predict graph constants, lexical labels and term types. We add an artificial root token to the beginning of the sequences of hidden states ($\mathbf{s}_0$ and $\mathbf{s'}_0$), which have their own learned embeddings.

The initial state of the decoder LSTM ($\mathbf{h}^{(0)}$) is the last hidden state of the encoder (forward component of $\mathbf{s}_n$ and backward component of $\mathbf{s}_1$) and is updated as follows

$$\mathbf{h}^{(t)} = \mathrm{LSTM}(\mathbf{h}^{(t-1)}, [\mathbf{s}_{tos}, \mathbf{s}_p, \mathbf{s}_c]),$$

where tos denotes the node active node on top of the stack, p the parent of tos and c refers to the most recently generated child of tos. This differs slightly from how Ma et al. (2018) take higher-order information into account because they sum those three vectors instead of concatenating them. Summing makes it harder for the model to keep track of the partially built tree because it cannot tell which of the three tokens plays which role. LTL builds the tree in a slightly different order (see Figure 3.6) and we found concatenation instead of summing crucial for predicting LTL transitions accurately.

Looking at the update rule of the decoder state, we now see that a separate Choose transition for LTF would not provide any new information to the neural network since it neither creates an edge nor changes the stack.

We compute the attention value $a_i^{(t)}$ exactly as defined by Ma et al. (Equation 2.1), which is repeated here for convenience:

$$\begin{aligned}
\mathbf{g}^{(t)} &= \mathrm{MLP}^{head}(\mathbf{h}^{(t)}) \\
\mathbf{r}_j &= \mathrm{MLP}^{dep}(\mathbf{s}_j) \\
e_j^{(t)} &= \mathbf{r}_j^T \mathbf{W} \mathbf{g}^{(t)} + \mathbf{U} \mathbf{g}^{(t)} + \mathbf{V} \mathbf{r}_j + \mathbf{b} \\
a_i^{(t)} &= \mathrm{softmax}(e^{(t)})_i
\end{aligned}$$

Furthermore, we compute probabilities for all labels $\ell \in Lab$, graph constants $G \in C$, types $\tau \in \Omega$ and lexical labels lex of interest:

$$\begin{aligned}
P_\theta(\ell|\mathbf{h}^{(t)}, tos \to j) &= \text{softmax}(\text{MLP}([\mathbf{h}^{(t)}, \mathbf{s}_j]))_\ell \\
P_\theta(G|\mathbf{h}^{(t)}) &= \text{softmax}(\text{MLP}([\mathbf{h}^{(t)}, \mathbf{s}'_{tos}]))_G \\
P_\theta(lex|\mathbf{h}^{(t)}) &= \text{softmax}(\text{MLP}([\mathbf{h}^{(t)}, \mathbf{s}'_{tos}]))_{lex} \\
P_\theta(\tau|\mathbf{h}^{(t)}) &= \text{softmax}(\text{MLP}([\mathbf{h}^{(t)}, \mathbf{s}'_{tos}]))_\tau
\end{aligned} \tag{4.2}$$

Note that we depart from Ma et al. (2018) and use simple feedforward networks instead of deep bi-affine classifiers. During development, we noticed that this leads to comparable performance but is considerably faster to compute.

4.3 Training

The training objective or training loss is to minimize the normalized negative log-likelihood of the training set D:

$$-\frac{1}{|D|}\sum_{i=1}^{|D|} \log P_\theta(d_i^{(1)}, \ldots, d_i^{(N)}|\mathbf{x_i})$$

where $d_i^{(1)}, \ldots, d_i^{(N)}$ is a gold transition sequence for sentence $\mathbf{x_i}$. The objective is minimized with gradient descent with an adaptive learning rate (see Appendix A for details). Following standard practice, we compute the loss only on a batch of the training set and update the parameters after every batch. A batch consists of a fixed number of sentences with similar length to maximize efficiency.

Since the training data is a corpus annotated with AM dependency trees, gold transition sequences need to be found. At this point, Algorithms 2 and 3 come in handy because this is exactly what they do.

There are usually multiple transitions sequences that lead to the correct AM dependency tree because the order in which children are added is arbitrary (see also Section 2.3). We follow Ma et al. (2018) and determine a canonical order of children that is "inside-out": first all children to the left of the head from right to left, then all children to the right of the head from left to right.

Note that the probability models presented above do not incorporate the constraints of the transition systems but are only incentivized to respect the constraints by the training data. This way, we can test to what degree the models learn the type constraints on their own. We enforce the constraints only at test time.

4.4 Inference

At test time, we use a version of the model that is constraint to valid transition sequences. We obtain it from the unconstrained models in Section 4.1 as follows:

$$Q_\theta(d^{(t)} = d|\mathbf{h}^{(t)}) = \frac{1}{Z}[\![d \text{ allowed}]\!] P_\theta(d^{(t)} = d|\mathbf{h}^{(t)})$$

where $[\![d \text{ allowed}]\!]$ sets the probability of ill-legal transitions according to LTF or LTL to 0. We do not need to compute the normalization constant Z to search for the most probable transition sequence. To see this, note that if d and d' are valid transitions after

Algorithm 6 Greedy inference for LTF with combined transitions

```
1: function $Greedy_{LTF}(Q_\theta, a^{(t)}, c)$
2:     $transition \leftarrow \emptyset$
3:     c' $\leftarrow c$
4:     if $\textsc{Choose}$ allowed in $c$ then
5:         $tr \leftarrow \textsc{Choose}(\arg\max_{\tau,G,lex} Q_\theta(\textsc{Choose}(\tau, G, lex)|c))$
6:         Let $c'$ be the result of applying $tr$ to $c$
7:     end if
8:     if $\arg\max_i a_i^{(t)} = 0$ and $\textsc{Pop}$ allowed in $c'$ then
9:         return $transition \oplus \textsc{Pop}$
10:    end if
11:    return $transition \oplus \arg\max_{o\in\{\textsc{Apply},\textsc{Modify}\},\alpha,i} Q_\theta(o(\alpha, i)|c)$
12: end function
```

$d^{(1)}, \dots, d^{(t)}$ and $P(d|\mathbf{h}^{(t)}) > P(d'|\mathbf{h}^{(t)})$ then $Q(d|\mathbf{h}^{(t)}) > Q(d'|\mathbf{h}^{(t)})$ because $\frac{1}{Z}$ is a constant factor.

In the following, we will leave the dependence on $\mathbf{h}^{(t)}$ implicit and write $Q_\theta(c)$ for the probability $Q_\theta(d^{(1)}, \dots, d^{(t)})$ where $d^{(1)}, \dots, d^{(t)}$ are the transitions that produced the configuration c.

Run time complexity Due to the unlimited influence of the transition history on the probability of the current transition, exact search is intractable, and we resort to greedy inference and beam search. For both LTF and LTL, a transition sequence for a sentence of length n has length $O(n)$. In every step, we have to compute $a^{(t)}$ which has to be computed over all n tokens. That is, run time complexity is $O(n^2)$ for greedy inference and there is another factor k (beam size) for beam search.

4.4.1 Greedy inference

The conceptually most straightforward way to perform greedy inference is to compute $\arg\max_d Q_\theta(d^{(t)} = d)$ at every step. However, this does not lead to the best performance in practice because of the factorization of P_θ. For example, if we use LTL and want to evaluate $Q_\theta(\textsc{Finish}(G, lex))$, we sometimes have the case that $a_0^{(t)}$ is very high, so the model is confident that we should perform *some kind* of $\textsc{Finish}(G, lex)$ but can be unsure about which graph constant G should be chosen here. Additionally, $\textsc{Finish}(G, lex)$ tends to have lower probability than Apply and Modify simply because it is the product of three distributions rather than two. A similar argument can be made for LTF, where Pop gets inappropriately high probabilities because it is modeled with the distribution over a single variable in contrast to Apply and Modify.

To reduce this kind of bias during inference, we first greedily decide whether to add a child or to stop adding children and then fill in the details. Algorithms 6 and 7 perform greedy inference for LTF with combined transition (Section 3.2.3) and LTL, respectively. As argument, they take the distribution Q_θ, the current attention value $a^{(t)}$ and the configuration of the transition system c and return the locally best transition.

We implement greedy inference on the CPU and proceed as follows: we compute (log) probabilities (Eq. 4.2), move them to the CPU, loop over all sentences in the batch, find the next transition and execute it. When this is done for all sentences in the batch, we perform the next step with the decoder network on the GPU to compute the next transition.

Algorithm 7 Greedy inference for LTL

```
1: function $Greedy_{LTL}(Q_\theta, a^{(t)}, c)$
2:     if $\arg\max_i a_i^{(t)} = 0$ and FINISH allowed in $c$ then
3:         return $\text{FINISH}(\arg\max_{G,lex} Q_\theta(\text{FINISH}(G, lex)|c))$
4:     end if
5:     return $\arg\max_{o\in\{\text{APPLY},\text{MODIFY}\},\alpha,i} Q_\theta(o(\alpha, i)|c)$
6: end function
```

4.4.2 Inference on GPUs

The inference algorithms (Algorithms 6 and 7) are easy to implement on a CPU but this results in relatively slow inference for two reasons: (i) scores have to be transferred from GPU memory to CPU memory and (ii) the procedure does not make use of the GPU to parallelize the computation across sentences. In this section, we focus on how we can implement greedy inference with LTL on GPUs.

While the computation on the GPU can be much faster, the implementation works differently. In particular, we have to compute batches of configurations and transitions, and the support for control flow is limited on GPUs. Essentially, all we have are indexing arrays, arithmetic and boolean operations, and comparisons (equality, greater than, etc.), which forces us to express the inference algorithm in this language.

The implementation on GPUs can be split into three major sub-problems:

(1) Representing a batch of configurations,

(2) Representing a batch of transitions,

(3) Updating a batch of configurations with a batch of transitions,

(4) Computing the best valid transitions for a batch of configurations

In order to focus on the essential parts, we assume that all sentences in the batch have the same length and that we use a transition sequence of the same length for all those sentences. In practice, this is obviously not the case, and we have to further incorporate padding the sentences and the transition sequence, which can be added straightforwardly.

Batches of configurations

The first step is to represent the configurations for an entire batch of sentences, which is shown in Table 4.1. We assume to be given a mapping that assigns each edge label to a unique natural number between 1 and $|Lab|$ so that we can easily represent an edge label with a number on the GPU and can reconstruct its identity later. We assume analogous mappings for graph constants and sources. `apply-set[b,i,s]` is true if and only if $s \in \mathbb{A}_c(i)$ for token `i` of sentence `b`. `stack` and `stack-ptr` together represent $\mathbb{S}$. We have to use a fixed-size array to represent the stack but this is not a problem because $\mathbb{S}$ grows at most linearly with the number of tokens in the sentence.

Note that we do not represent the set of term types explicitly in Table 4.1. When needed, we can derive it easily because it is fully determined by the lexical type of the parent of the current node and the label of the incoming edge (see definition of $PossT$).

The two elementary operations of a stack – push and pop – are described in Algorithms 8 and 9. Each argument is annotated with its shape. What makes these implementations a bit special is that they take an additional `mask` argument, which is a bool vector that tells us for each element in the batch if we actually want to push (or pop) or if we want

Name	Shape	Data type	Description
`stack`	$b \times n$	long	Represents a stack of max. depth n
`stack-ptr`	b	long	Points to current top elements in `stack`
`heads`	$b \times n$	long	Maps each token to the position of its head
`edge-labels`	$b \times n$	long	Maps each token to the label of the incoming edge
`constants`	$b \times n$	long	$\mathbb{G}$, maps each token to its graph constant
`apply-set`	$b \times n \times \lvert\mathcal{S}\rvert$	bool	$\mathbb{A}$, maps each token to its apply set
`w`	b	long	W_c

Table 4.1: Representation of a batched configuration. b refers to batch size, n is the sentence length and $\mathcal{S}$ are the sources that occur in our graph lexicon.

Algorithm 8 Push for batched stack

```
1: function PUSH(stack:(b × n), stack-ptr:(b), vector:(b), mask:(b))
2:     stack-ptr[b] += mask[b]
3:     keep[b] = (1-mask[b]) * stack[b, stack-ptr[b]]
4:     stack[b, stack-ptr[b]] = keep[b] + mask[b] * vector[b]
5: end function
```

this to have no effect. If we call push and then pop with different masks, we can effectively push a token to the stack for some sentences in the batch and pop the stack for others.

Batches of transitions

In LTL, transitions are parameterized by tokens (target of an edge), edge labels and graph constants, so all those objects have to be represented in the GPU memory; Table 4.2 shows how we do this. Since we batch transitions, we have to represent for each batch element, which kind of transition this is. We do this with a mask where `finish-mask[b]=1` means that we perform FINISH in `b`; otherwise we perform an edge creating transition whose identity is determined by `edge-labels[b]`.

Updating a batch of configurations

Given a batch of configurations and a batch of transitions, we want to update our configurations using these transitions to go forward in the derivation. The first thing we do is to determine which tokens are currently active, i.e. which tokens are on top of the stack:

`active[b] = stack[b, stack-ptr[b]]`

We can then update the arrays representing the configurations one by one. Let us consider the case of `heads`, to which we want to add the information about new edges and which we update using Algorithm 10 as UPDATE(`heads, selected, (1 - finish-mask), active`). Here, `heads` plays the role where the information is stored, `selected` directs where the new information is added (namely we modify the heads of the tokens that were selected), `1 - finish-mask` identifies the batch elements where actually something is updated and `active` provides the new information, namely that we draw the edges *from* the nodes that are currently on top of the stack. The same or very similar update principles apply also for `edge-labels`, `constants` and `apply-set`. The number of words without incoming edges is updated by setting `w[b] -= (1 - finish-mask[b])`, since the cases where we do not FINISH are exactly those where a token receives an incoming edge.

Finally, we have to update the stack. In LTL, the FINISH transition pops the stack

Algorithm 9 Pop for batched stack

```
1: function POP(stack:(b × n), stack-ptr:(b), mask:(b))
2:     top[b] ← stack[b, stack-ptr[b]]
3:     stack-ptr[b] = stack-ptr[b] - mask[b]
4:     return top
5: end function
```

Name	Data type	Description
`selected`	long	Endpoints of edges
`finish-mask`	bool	On which sentences to perform FINISH
`edge-labels`	long	Which edge labels were selected
`constants`	long	Which constants were selected

Table 4.2: Representation of the transitions at a certain time step for a batch of sentences. All arrays are vectors with one entry per element in the batch.

and then pushes all children of the active node. We have to make sure that we pop only for those batch elements where we actually perform FINISH, which is achieved by `POP(stack, stack-ptr, finish-mask)`. We then collect the children of the active nodes and repeatedly PUSH until they are all on the stack.

Computing best valid transitions

The most important and most intricate part is computing the best valid transitions given a batch of configuration. At step t, we get three arrays with probabilities from the neural network: `scores` of shape $b \times n$, `label-scores` of shape $b \times n \times |Lab|$, and `constant-scores` of shape $b \times |C|$, which contain $a^{(t)}$, $P(\ell = \cdot | \mathbf{h}^{(t)}, tos \to \cdot)$ and $P(G = \cdot | \mathbf{h}^{(t)})$ respectively. We cannot simply choose the label or graph constant with the highest probability because that might result in a transition that flouts the rules of LTL. Instead, let us assume that we had the following binary arrays:

- `finish` of shape $b \times C$, indicating for all sentences in the batch which FINISH$(G), G \in C$ are allowed.
- `label-mask` of shape $b \times Lab$ indicating for all sentences, if we are allowed to perform an APPLY or MODIFY transition that creates an edge with the respective label.

If we had those arrays, we could "mask out" scores of invalid choices, that is, we could multiply the mask with the respective score array, setting invalid choices to 0. *Then* we can select the highest scoring graph constant or label. Sometimes, it will be the case that `finish[b,c]=0` for all graph constants `c`, which means that no FINISH transition can be applied at all. If this happens, we also mask out `scores[b,0]` and make sure to set `finish-mask[b]=0` in the transition. Consequently, an edge will be drawn instead (recall that we model FINISH with selecting the token in position 0).

Let us now look into how we can actually compute `finish` and `label-mask`. To save time during parsing we pre-compute several arrays and store them in the GPU memory (see Table 4.3). From the batched configuration, we can compute the following arrays:

- `curr-lex-types` of shape b, containing the lexical types of the parents of the active nodes,

Algorithm 10 Update information in array

```
1: function UPDATE(array:(b,n), where:(b), if:(b), new:(b) )
2:     array[b,where[b]] = (1-if[b]) * array[b, where[b]] + if[b] * new[b]
3: end function
```

Name	Shape	Definition
ALL-APPLYSETS	$\lvert\Omega\rvert \times \lvert\Omega\rvert \times \lvert\mathcal{S}\rvert$	ALL-APPLYSETS$[\tau, \lambda, s] = 1$ iff $s \in \mathcal{A}(\lambda, \tau)$
REACHABLE	$\lvert\Omega\rvert \times \lvert\Omega\rvert$	REACHABLE$[\tau, \lambda] = 1$ iff τ apply reachable from λ
POSST	$\lvert\Omega\rvert \times \lvert Lab\rvert \times \lvert\Omega\rvert$	POSST$[\lambda, \ell, \tau] = 1$ iff $\tau \in PossT(\lambda, \ell)$
L2C	$\lvert\Omega\rvert \times \lvert C\rvert$	L2C$[\lambda, G] = 1$ iff $\tau(G) = \lambda$
S2L	$\lvert\mathcal{S}\rvert \times \lvert Lab\rvert$	S2L$[s, \ell] = 1$ iff $\ell = \text{APP}_s$
IS-MOD	$\lvert Lab\rvert$	IS-MOD$[\ell] = 1$ iff $\ell = \text{MOD}_\beta$, for some β

Table 4.3: Pre-computed bool arrays needed for type checking.

- `applyset` of shape $b \times \mathcal{S}$, representing $\mathbb{A}$ for the active nodes, and
- `incoming-labels` of shape b with the labels of edges pointing to the active nodes.

With these arrays, we can compute the set of term types of the active nodes: `term-types[b,t]` `=` POSST`[curr-lex-types[b], incoming-labels[b], t]`

Using the information about the set of term types of the active tokens (`term-types`), which outgoing APP edges have been drawn already (`applyset`) and how many tokens there are left (`w`), Algorithm 11 computes the two arrays `finish` and `edge-mask`. When looking at Algorithm 11, it is useful to bear in mind how some logical statements can be expressed in arithmetic terms: a conjunction $P \wedge Q$ is a multiplication `P * Q` and sometimes it is more efficient to express a disjunction $\bigvee_{\texttt{x}}$ `P[x]` as $(\sum_{\texttt{x}}$ `P[x]) > 0`.

First, we want to compute which pairs (τ, λ) are consistent with the sources that we have collected so far in `applyset`. In other words, we want to compute $\{(\tau, \lambda) | \tau \in \mathbb{T}(tos) \wedge \mathbb{A}(tos) \subseteq \mathcal{A}(\lambda, \tau)\}$. The way we check the subset-relation is by the following trick: $X \subseteq Y$ iff $|X \cap Y| = |X|$. We first compute the `overlap`, that is $|\mathbb{A} \cap \mathcal{A}(\lambda, \tau)|$. The term ALL-APPLYSETS`[t,l,s] * applyset[b,s]` is 1 iff we have collected source `s` *and* `s` is in the apply set $\mathcal{A}(\texttt{l}, \texttt{t})$. By summing over all sources, we compute the entire overlap. For type pairs (τ, λ) where τ is not apply reachable from λ, the `overlap` is 0, which is indistinguishable from the case that the apply set is empty. We introduce this distinction by setting `overlap` to $-\infty$ for those pairs. We also have to take the constraint into account that $\tau \in \mathbb{T}(tos)$, which we do by setting `overlap` to $-\infty$ for all $\tau \notin \mathbb{T}(tos)$. To compute `consistent`, we then only have to check for which combination of λ, τ, it holds that $|\mathbb{A}(tos)| = |\mathbb{A}(tos) \cap \mathcal{A}(\lambda, \tau)|$. Note that setting `overlap` to a negative number for a pair λ, τ means that the above equality cannot hold because $|\mathbb{A}(tos)|$ is a natural number.

`Finish` The FINISH transition can be applied exactly in the case where we can provide a graph constant G and a term type $\tau \in \mathbb{T}(tos)$ such that $\mathcal{A}(\tau(G), \tau) = \mathbb{A}(tos)$ (see Section 3.3). Note that equality on sets can be rephrased as $\mathbb{A}(tos) \subseteq \mathcal{A}(\tau(G), \tau) \wedge \mathcal{A}(\tau(G), \tau) \subseteq \mathbb{A}(tos)$. We have computed the first part already (`consistent`), and can compute the second part analogously, resulting in an array `finish`, where `finish[b,t,l]=1` means that the equality holds for the lexical type with id `l` and term type `t`.

To compute the graph constants which satisfy the requirements of FINISH we first calculate all lexical types that make FINISH applicable. We can do this by a disjunction

Algorithm 11 Compute allowed transitions on GPU

```
1: function TRANSITION-MASKS(term-types:(b $\times \Omega$), applyset:(b $\times \mathcal{S}$), w:(b))
2:     overlap[b,t,l] = $\sum_s$ ALL-APPLYSETS[t,l,s] * applyset[b,s]
3:     overlap[b,t,l] += $-\infty$ * $(1 -$ REACHABLE[t,l]$)$
4:     overlap[b,t,l] += $-\infty$ * $(1 -$ term-types[b,t]$)$
5:
6:     applyset-size[b] = $\sum_s$ applyset[b,s]
7:     consistent[b,t,l] = overlap[b,t,l] == applyset-size[b]
8:
9:     finish[b,t,l] = overlap[b,t,l] == $\sum_s$ ALL-APPLYSETS[t,l,s]
10:    finish[b,t,l] *= consistent[b,t,l]
11:    finish[b,l] = $\bigvee_{t}$ finish[b,t,l]
12:    finish[b,c] = ($\sum_l$ finish[b,l] * L2C[l,c]) > 0
13:
14:    todo[b,t,l] = $\sum_s$ ALL-APPLYSETS[t,l,s] - applyset-size[b]
15:    consistent[b,t,l] *= todo[b,t,l] $\leq$ w[b]
16:    app-mask[b,s] = $\bigvee_{t,l}$consistent[b,t,l] * ALL-APPLYSETS[t,l,s]
17:    app-mask[b,s] *= (1-applyset[b,s])
18:
19:    todo[b,t,l] += $\infty$ * (1-consistent[b,t,l])
20:    o[b] = min$_{t,l}$ todo[b,t,l]
21:    mod-mask[b] = (w[b] - o[b]) $\geq$ 1
22:
23:    label-mask[b,l] = ($\sum_s$ app-mask[b,s] * S2L[s,l]) > 0
24:    label-mask[b,l] += IS-MOD[l] * mod-mask[b]
25:
26:    return finish, label-mask
27: end function
```

over the term types to obtain `finish[b,l]`. `finish[b,l]=1` implies that there is indeed some term type $\tau \in \mathbb{T}(tos)$ such that $\mathcal{A}(\mathtt{l}, \tau) = \mathbb{A}(tos)$. The only thing left to do is to translate this set of lexical types to a set of graph constants, that is, finding all graph constants with id `c` whose lexical type is `l` (see L2C) for which `finish[b,l]` holds (line 12). Note that this can be done efficiently by a matrix multiplication and a comparison (Line 12).

`Label-mask` We first focus on determining for which *sources* we can perform APPLY$(\alpha, \cdot)$ in the current configuration. Then we will look at MODIFY and combine the information into the `label-mask`.

The pre-condition for APPLY$(\alpha, \cdot)$ requires that there is some term type $\tau \in \mathbb{T}(tos)$ such that $PossL(\tau, \mathbb{A}(tos) \cup \{\alpha\}, W - 1)$ is non-empty. Expanding this condition, APPLY$(\alpha, \cdot)$ is allowed for the following sources:

$$\{\alpha \in \mathcal{S} \,|\, \exists \tau \in \mathbb{T}(tos), \lambda \in \Omega.\, \mathbb{A}(tos) \cup \{\alpha\} \subseteq \mathcal{A}(\lambda, \tau) \wedge |\mathcal{A}(\lambda, \tau) - (\mathbb{A}(tos) \cup \{\alpha\})| \leq W - 1\}$$

which is equivalent to

$$\{\alpha \in \mathcal{S} \mid \exists \tau \in \mathbb{T}(tos), \lambda \in \Omega.\, \mathbb{A}(tos) \subseteq \mathcal{A}(\lambda, \tau) \wedge \alpha \in \mathcal{A}(\lambda, \tau) \wedge |\mathcal{A}(\lambda, \tau) - \mathbb{A}(tos)| \leq W\}$$

since $\mathbb{A}(tos) \subseteq \mathcal{A}(\lambda, \tau)$, we can choose the computationally more convenient form

$$\{\alpha \in \mathcal{S} \mid \exists \tau \in \mathbb{T}(tos), \lambda \in \Omega.\ \mathbb{A}(tos) \subseteq \mathcal{A}(\lambda,\tau) \wedge \alpha \in \mathcal{A}(\lambda,\tau) \wedge |\mathcal{A}(\lambda,\tau)| - |\mathbb{A}(tos)| \leq W\} \tag{4.3}$$

In order to compute this, we will first compute the set $\{(\lambda, \tau) \mid \tau \in \mathbb{T}(tos) \wedge \mathbb{A}(tos) \subseteq \mathcal{A}(\lambda,\tau) \wedge |\mathcal{A}(\lambda,\tau)| - |\mathbb{A}(tos)|\}$. Except for the last condition, this is exactly what we stored in `consistent`. We create a new array `todo[b,t,l]`, which contains $|\mathcal{A}(\mathtt{l},\mathtt{t})| - |\mathbb{A}(tos)|$, that is, it stores for `t` and `l`, how many sources we still have to fill for this combination to type check. Next, we restrict `consistent` to those pairs `t`, `l` where the number of sources we still have to fill is at most the number of words that are left in the sentence (Line 15). Finally, we can compute the set in Eq. 4.3 by considering all those sources `s` such that there exists a pair `t`, `l` that is `consistent` and where `s` is in the apply set from $\mathcal{A}(\mathtt{l},\mathtt{t})$.

The transition $\textsc{Apply}(\alpha, \cdot)$ also has the pre-condition $\alpha \notin \mathbb{A}(tos)$, which is easily taken into account as well (Line 17).

Next, we investigate for all sentences in the batch if $\textsc{Modify}$ is allowed. The only pre-condition for $\textsc{Modify}$ is that $W_c - O_c \geq 1$. While W_c is known (`w`), we have to compute O_c. For LTL, it can be shown that $O_c = \sum_i O_c(i) = O(tos)$, if the top of stack, *tos*, is defined. Intuitively, this comes from the fact that we always work on one token at a time and only continue with the next one after $\textsc{Finish}$ing the old one. Since *tos* is defined in all interesting cases, we have to compute

$$O_c(tos) = \min_{\lambda \in \Omega, \tau \in \mathbb{T}_c(i)} |\mathcal{A}(\lambda,\tau) - \mathbb{A}_c(i)|$$

We are only interested in the case where λ, τ are consistent with the apply edges that we have drawn ($\mathbb{A}_c(i) \subseteq \mathcal{A}(\lambda,\tau)$), which simplifies $O_c(tos)$ to

$$O_c(tos) = \min_{\lambda \in \Omega, \tau \in \mathbb{T}_c(i)} |\mathcal{A}(\lambda,\tau)| - |\mathbb{A}_c(i)|$$

Note that this quantity basically is what is stored in `todo`. However, we have to account for pairs `l`, `t` that are not `consistent`, and set those to ∞ so they will not influence the minimum (Line 19).

Finally, we can compute `label-mask` which tells us what edge labels are valid in the current configuration. In Line 23, we translate the information in `app-mask`, which contains the information which *source names* we are allowed to use in $\textsc{Apply}$, into information about *edge labels*. If `mod-mask[b]=1`, we can draw *any* $\textsc{mod}$ edge.

4.4.3 Beam search

Greedy inference is fast and often sufficiently accurate but beam search improves accuracy by pursuing more than just one option. Since the search space grows exponentially with the depth of the derived tree, it is not possible to explore everything. Beam search limits itself to the k most likely partial parses. We also say that the beam size is k.

Algorithm 12 shows the beam search procedure we use. It receives as input the distribution Q_θ, the length of the sentence n and the beam size k. In the algorithm, let $top^k_{a \in A} f(a)$ be the function that returns the k largest elements of $A' = \{a \in A \mid f(a) > 0\}$ according to f. If A' has fewer elements than k, it returns A'. Where A is obvious, we leave it implicit.

Algorithm 12 Beam search

```
1: function BEAM-SEARCH($Q_\theta, n, k$)
2:     $frontier \leftarrow \{\langle\emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle\}$
3:     for $t$ in $1 \ldots 2n$ do
4:         $candidates \leftarrow \emptyset$
5:         for all $c \in frontier$ do
6:             $candidates \leftarrow candidates \cup \{(tr, c) \mid tr \in \text{GET-CANDIDATES}(Q_\theta, a^{(t)}, c, k)\}$
7:         end for
8:         $frontier \leftarrow \emptyset$
9:         for all $t, c' \in top^k_{(tr,c') \in candidates} Q_\theta(tr) \cdot Q_\theta(c')$ do
10:            if $c'$ is a goal configuration then
11:                $frontier \leftarrow frontier \cup \{c'\}$
12:            else
13:                $frontier \leftarrow frontier \cup \{t \text{ applied to } c'\}$
14:            end if
15:        end for
16:    end for
17:    return $\arg\max_{c \in frontier} Q_\theta(c)$
18: end function
```

Algorithm 13 Get LTL candidates

```
1: function GET-CANDIDATES$_{LTL}$($Q_\theta, a^{(t)}, c, k$)
2:     $candidates \leftarrow top^k\, Q_\theta(\text{FINISH}(G)|c)$
3:     for $ch \in top^k_i\, a^{(t)}_i$ do
4:         $candidates \leftarrow candidates \cup top^k\, Q_\theta(\text{APPLY}(\alpha, ch)|c)$
5:         $candidates \leftarrow candidates \cup top^k\, Q_\theta(\text{MODIFY}(\beta, ch)|c)$
6:     end for
7:     return $top^k_{d \in candidates} Q_\theta(d|c)$
8: end function
```

In each step of Algorithm 12, we generate (up to) k possible transitions for all configurations in the frontier and only keep those k with the highest probability. The sub-routine GET-CANDIDATES returns the k most likely valid transitions according to either LTF or LTL. We do not have to look at more than the k most likely transitions for every configuration because, in the extreme case, exactly those k transitions will fill the frontier, anything beyond that will be discarded anyway.

Algorithm 13 computes the k most likely next transitions for LTL in a straightforward way. In writing $Q_\theta(\text{FINISH}(G)|c)$, we mean that this does *not* include the term that assigns probability to a lexical label because this biases the beam search towards inappropriately delaying FINISH. Note that the lexical label has no influence on how the search proceeds, and we choose it greedily.

Algorithm 14 computes the k most likely next transitions for LTF. Since we use the variant of LTF with combined transitions, we make a distinction between the case where we have to predict CHOOSE combined with another transition and the case where we predict a single transition. If we have to CHOOSE, we consider the k best CHOOSE transitions and apply them to the current configuration c. For every possible configuration that we might be in then, we generate the best k possible transitions for every transition type. Finally, we return the k highest scoring combined transitions. The case where we do not have to use a CHOOSE transition is analogous to Algorithm 13.

Algorithm 14 Get LTF candidates

```
function GET-CANDIDATES$_{LTF}(Q_\theta, a^{(t)}, c, k)$
    $candidates \leftarrow \emptyset$
    if CHOOSE allowed in $c$ then
        $Tr_{choose} \leftarrow top^k\ Q_\theta(\text{CHOOSE}(\tau, G)|c)$
        for $tr \in Tr_{choose}$ do
            $c' \leftarrow tr$ applied to $c$
            for $ch \in top_i^k\ a_i^{(t)}$ do
                $candidates \leftarrow candidates \cup \{(tr, tr') \mid tr' \in top^k\ Q_\theta(\text{APPLY}(\alpha, ch)|c')\}$
                $candidates \leftarrow candidates \cup \{(tr, tr') \mid tr' \in top^k\ Q_\theta(\text{MODIFY}(\beta, ch)|c')\}$
            end for
            if POP allowed in $c'$ then
                $candidates \leftarrow candidates \cup \{(tr, \text{POP})\}$
            end if
        end for
        return $top^k_{d_1, d_2 \in candidates} Q_\theta(d_1, d_2|c)$
    end if
    for $ch \in top_i^k\ a_i^{(t)}$ do
        $candidates \leftarrow candidates \cup top^k\ Q_\theta(\text{APPLY}(\alpha, ch)|c)$
        $candidates \leftarrow candidates \cup top^k\ Q_\theta(\text{MODIFY}(\beta, ch)|c)$
    end for
    if POP allowed in $c$ then
        $candidates \leftarrow candidates \cup \{\text{POP}\}$
    end if
    return $top^k_{d \in candidates} Q_\theta(d|c)$
end function
```

Chapter 5

Experiments

We investigate two important empirical questions in this chapter:

1. How accurate is AM dependency parsing with the LTF and LTL transition systems in practice, and how does that compare to existing AM dependency parsers and to entirely different approaches to semantic parsing?

2. How fast are LTF and LTL in practice, and how does that compare to the existing parsing algorithms for AM dependency trees?

We approach this by implementing the parsing algorithms and models described in chapter 3 and 4, training them on AM dependency trees, evaluating speed and parsing accuracy on unseen test data, and compare these to the results obtained by Lindemann et al. (2019) (L'19 for short).

We evaluate all models in two conditions: with traditional non-contextualized GloVe embeddings (Pennington et al., 2014) and with the contextualized BERT embeddings (Devlin et al., 2019).

5.1 Data

The parsing model of chapter 4 is trained on AM dependency trees and on a theoretical level is agnostic towards what kind of semantic graph the AM dependency trees represent. In order to get a clear picture of strengths and weaknesses in practice, we perform experiments on several graphbanks, namely DM, PAS, PSD, EDS and two versions of the AMR bank LDC2015E86 ("AMR 2015") and LDC2017T10 ("AMR 2017"), that differ in the amount of available training data. We use the AM dependency trees that L'19 extracted from the training sets of these graphbanks.

We follow the standard splits into training, development and test data specific to the graphbanks. DM, PAS, PSD and EDS are annotated on the same corpus of financial news, the Wall Street Journal section of the Penn Tree Bank (Marcus et al., 1993), whereas AMR is annotated on a more diverse set of genres, ranging from news to online forums and blog posts.

While there is only one test set for EDS and AMR, there are two for the other graphbanks: an in-domain test set stemming from the Wall Street Journal and an out-of-domain test set, stemming from the Brown section of the Penn Tree Bank.

Table 5.1 shows the number of sentences and AM dependency trees in the different data splits. The EDS training set is smaller than that of DM, PAS and PSD because we only consider the subset of the training set where graphs are connected.

	Training		**Development**		**Test** (id)		**Test** (ood)	
	Sent.	AM dep. trees	Sent.	AM dep. trees	Sent.	Tokens	Sent.	Tokens
DM	35,657	31,349	1,692	1,577	1,410	33,358	1,849	33,432
PAS	35,657	31,796	1,692	1,661	1,410	33,358	1,849	33,432
PSD	35,657	32,807	1,692	1,638	1,410	33,358	1,849	33,432
EDS	33,964	25,680	1,692	1,000	1,410	32,306	-	-
AMR 15	16,833	15,472	1,367	1,240	1,371	28,458	-	-
AMR 17	36,521	33,406	1,367	1,214	1,371	28,458	-	-

Table 5.1: Number of sentences, AM dependency trees and tokens after preprocessing.

Graphbank	**Constants**	**Ω**
DM	429	102
PAS	160	85
PSD	1643	337
EDS	2144	144
AMR 15	2565	284
AMR 17	3597	331

Table 5.2: Number of graph constants and size of Ω per graphbank.

Note that the heuristic method of L'19 does not produce AM dependency trees for the entire training set. No data from the test sets was excluded. We refer to the part of the development set for which there are AM dependency trees as the *decomposable development set.*

Table 5.2 shows for each graphbank how many delexicalized graph constants were extracted by L'19 and how large the set of all types is.

5.2 Setup

This section provides details about the hardware we used and pre- and post-processing decisions. The hyperparameters we used can be found in Appendix A.

Environment All models are implemented with PyTorch 1.4 and AllenNLP 0.9. We trained on Nvidia Tesla V100 graphics cards and Intel Xeon Gold 6128 CPUs running at 3.40 GHz. This was also the setup for parsing experiments.

Evaluation metrics We use variants of labeled F-score to evaluate the predictions of our models. For DM, PAS, PSD, F-score can be computed exactly because the nodes of the graph are anchored in the tokens in the sentence. For AMR and EDS, we evaluate with Smatch (Cai and Knight, 2013), which approximates the F-score because the nodes are not anchored in the sentence, leading to an NP-complete problem. Finally, we also use the EDM metric for EDS (Dridan and Oepen, 2011), which not only measures the overlap between gold graph and predicted graph but also takes into account if the anchoring was predicted correctly; thus it is stricter than Smatch.

Pre- and post-processing We re-use the entire pre- and post-processing pipeline of L'19. That is, if gold tokenization, lemmas and POS tags are given (DM, PAS, PSD), we use those. For EDS and AMR, we tokenize, tag and lemmatize with CoreNLP (Manning et al., 2014) and use L'19's refinement rules, e.g. attaching punctuation to the end of the last word of the sentence for EDS. We tag all corpora using CoreNLP for named entities.

The graphs of DM, PAS and PSD are not necessarily connected. To cope with that, we follow L'19 in adding a token to the end of the sentence that has outgoing edges to some node in every connected component.

The nodes of EDS graphs are anchored in spans of the sentence, which can comprise words or even entire phrases. Ideally, one would use a separate component to predict this anchoring. For simplicity, we use L'19's heuristic method to compute them. If we apply this post-processing on the graphs we obtain from evaluating the AM dependency trees in the training set and compare them to the gold graphs (round-trip conversion), we achieve an accuracy of 89.7 EDM F-score.

Finding AM dependency trees for AMR involves aligning them to the tokens in the sentence. The pipeline of L'19 uses a rule-based aligner (Groschwitz et al., 2018). There was in bug in the post-processing of AMR that negatively impacted the published results of L'19. All results reported here use a version of the post-processing where the bug has been fixed.

Additional graph constants The guarantee that the parser cannot run into dead ends depends on a few crucial assumptions (see Theorem 3.4.9). These assumptions are almost perfectly met in practice. In the worst case (PSD), we have to add 14 graph constants to make them true, which increases the size of the graph lexicon by less than 1%. The graph constants that we add are automatically constructed from their types: they consist of one node per source. Additionally, there is a central node from which edges point to the nodes with the sources.

Parsing experiments We use a batch size of 64 for parsing experiments where the type constraints are computed on the CPU. If they are computed on the GPU, we use a batch size of 512, except for AMR, for which we use a batch size of 256 because of limited GPU memory. Results reported for the projective parser and the fixed-tree parser use the 6 highest scoring graph constants ("supertags"). When parsing with the fixed tree parser is not completed with k supertags within 30 minutes, we retry with $k-1$ supertags. If $k=0$, we return a dummy graph. When we parse with the projective parser, we skip the two longest sentences in the AMR test sets and use dummy graphs as well.

Models for projective parser We also compare the performance to using the projective parser (see Section 2.2.2), which L'19 did not use for evaluation. We compute scores using re-trained models of L'19 but with the difference that we use the edge existence loss recommended by Groschwitz et al. (2018) since this resulted in better performance in their experiments. We select the best epoch based on the arithmetic mean of labeled attachment score and supertagging accuracy on the decomposable development set because it is computationally too expensive to parse the development set after every epoch.

5.3 Accuracy

We train four models with different random initialization per transition system and graphbank and report averages and standard deviations of the F-scores. Table 5.3 shows the results on the test sets for LTL and LTF with greedy search and beam search in

	DM		PAS		PSD		EDS		AMR 15	AMR 17
	id F	ood F	id F	ood F	id F	ood F	Smatch F	EDM	Smatch F	Smatch F
L'19, projective	88.8±0.1	83.2±0.2	91.7±0.2	87.1±0.2	77±0.1	74.4±0.1	85.6±0.1	80.8±0.1	69.1±0.1	70.0±0.1
L'19, fixed tree	90.4±0.2	84.3±0.2	91.4±0.1	86.6±0.1	78.1±0.2	74.5±0.2	87.6±0.1	82.5±0.1	70.0±0.1	71.2±0.1
L'19, fixed tree+CNN	90.5±0.1	84.5±0.1	91.5±0.1	86.5±0.1	78.4±0.2	74.8±0.2	87.7±0.1	82.8±0.1	70.2±0.4	71.4±0.2
LTL, greedy, -types	85.6±0.4	75.5±0.2	86.0±0.9	77.8±0.5	59.6±1.1	54.9±0.3	77.9±0.4	74.0±0.4	34.0±1.3	39.0±2.0
LTL, greedy	91.4±0.2	85.4±0.1	92.5±0.1	87.9±0.1	78.6±0.2	74.9±0.3	88.2±0.1	83.0±0.1	70.8±0.2	72.8±0.3
beam = 3	**91.5**±0.2	**86.0**±0.1	**92.7**±0.2	**88.3**±0.3	**79.4**±0.2	**76.2**±0.2	**88.3**±0.2	**83.1**±0.1	**71.4**±0.3	**73.5**±0.3
LTF, greedy, -types	80.1±1.3	69.5±0.4	83.5±0.3	73.8±1.0	56.5±1.0	50.8±1.1	69.4±0.6	66.3±0.6	25.8±1.0	29.6±0.9
LTF, greedy	89.7±0.4	83.0±0.3	91.8±0.2	86.6±0.2	74.2±0.4	69.8±0.6	86.1±0.1	81.3±0.1	67.5±0.2	69.0±0.3
beam = 3	**91.5**±0.3	85.6±0.2	92.6±0.2	88.0±0.1	78.8±0.4	75.1±0.2	88.1±0.2	82.9±0.1	70.7±0.2	72.0±0.2
L'19, projective	91.6±0.1	88.2±0.2	94.4±0.1	92.6±0.1	81.6±0.1	81.5±0.2	87.5±0.6	82.8±0.1	74.2±0.1	75.1±0.1
L'19, fixed tree	**93.9**±0.1	90.3±0.1	94.5±0.1	92.5±0.1	**82.0**±0.1	81.5±0.3	90.1±0.1	84.9±0.1	75.1±0.1	76.0±0.2
L'19, fixed tree+CNN	93.8±0.1	90.2±0.1	94.6±0.1	92.5±0.1	81.9±0.1	81.5±0.2	90.2±0.1	85.0±0.1	75.1±0.2	76.1±0.1
LTL, greedy, -types	88.5±0.3	82.9±0.4	88.3±0.8	83.6±0.9	67.2±0.6	67.3±0.7	80.5±0.2	76.3±0.2	39.5±0.5	46.9±1.0
LTL, greedy	93.7±0.2	90.0±0.1	94.6±0.2	92.5±0.2	81.4±0.2	80.7±0.2	90.2±0.1	85.0±0.0	74.6±0.3	76.3±0.1
beam=3	**93.9**±0.1	90.4±0	**94.7**±0.1	**92.7**±0.2	81.9±0.1	**81.6**±0.1	**90.4**±0.0	**85.1**±0.0	**75.4**±0.3	**76.8**±0.1
LTF, greedy, -types	85.0±0.3	78.0±0.7	86.9±0.6	81.4±0.4	63.1±1.4	62.5±0.7	72.4±0.4	69.1±0.3	30.8±0.3	36.6±2.7
LTF, greedy	92.5±0.1	88.4±0.2	94.0±0.2	91.5±0.2	77.7±0.4	76.5±0.5	88.0±0.3	83.0±0.3	71.2±0.2	72.9±0.4
beam=3	**93.9**±0.1	**90.5**±0.1	94.6±0.2	92.6±0.1	81.3±0.1	80.8±0.1	90.0±0.1	84.8±0.1	74.6±0.1	75.9±0.2

Table 5.3: Semantic parsing accuracies (id = in-domain test set; ood = out-of-domain test set) in comparison to existing AM dependency parsers. Models above the line use GloVe, models below the line use BERT. L'19 is Lindemann et al. (2019) without multi-task learning (incl. post-processing bugfix).

	DM		PAS		PSD		EDS		AMR 15	AMR 17
	id F	ood F	id F	ood F	id F	ood F	Smatch F	EDM	Smatch F	Smatch F
Dozat and Manning (2018)	93.7	88.9	94.0	90.8	81.0	79.4	-	-	-	-
Wang et al. (2019)	**94.0**	**89.7**	94.1	**91.3**	81.4	79.6	-	-	-	-
FG'20	93.9	89.6	**94.2**	91.2	**81.8**	**79.8**	-	-	-	-
Lyu and Titov (2018)	-	-	-	-	-	-	-	-	**73.7**	**74.4**±0.16
Chen et al. (2018)	-	-	-	-	-	-	**90.9**	**90.4**	-	-
L'19, projective	88.8±0.1	83.2±0.2	91.7±0.2	87.1±0.2	77±0.1	74.4±0.1	85.6±0.1	80.8±0.1	69.1±0.1	70.0±0.1
L'19, fixed tree	90.4±0.2	84.3±0.2	91.4±0.1	86.6±0.1	78.1±0.2	74.5±0.2	87.6±0.1	82.5±0.1	70.0±0.1	71.2±0.1
LTL, beam = 3	91.5±0.2	86.0±0.1	92.7±0.2	88.3±0.3	79.4±0.2	76.2±0.2	88.3±0.2	83.1±0.1	71.4±0.3	73.5±0.3
He and Choi (2020)	**94.6**	90.8	**96.1**	**94.4**	**86.8**	79.5	-	-	-	-
FG'20	94.4	**91.0**	95.1	93.4	82.6	**82.0**	-	-	-	-
Cai and Lam (2020)	-	-	-	-	-	-			-	**80.2**
Zhang et al. (2019)	92.2	87.1	-	-	-	-	-	-	-	77.0±0.1
L'19, projective	91.6±0.1	88.2±0.2	94.4±0.1	92.6±0.1	81.6±0.1	81.5±0.2	87.5±0.6	82.8±0.1	74.2±0.1	75.1±0.1
L'19, fixed tree	93.9±0.1	90.3±0.1	94.5±0.1	92.5±0.1	82.0±0.1	81.5±0.3	90.1±0.1	84.9±0.1	75.1±0.1	76.0±0.2
LTL, beam=3	93.9±0.1	90.4±0.0	94.7±0.1	92.7±0.2	81.9±0.1	81.6±0.1	90.4±0.0	85.1±0.0	**75.4**±0.3	76.8±0.1

Table 5.4: Comparison to state-of-the-art semantic parsing accuracies. All models below the line use BERT. FG'20 is Fernández-González and Gómez-Rodríguez (2020)

comparison to the existing AM dependency parsers (projective, and fixed tree) that use the model of L'19. The comparison to L'19 is potentially unfair since these models do not have direct access to character information via a character CNN. In order to exclude this confound, we re-train the fixed-tree models with a character CNN using the same hyperparameters as used for LTF and LTL.

GloVe When using GloVe embeddings, LTL consistently outperforms L'19 with greedy parsing but the gap becomes even larger with beam search. For AMR 17, LTL outperforms L'19 by more than 2 points F-score.

Results for LTF with greedy parsing are mixed; for PAS it works on-par with L'19 but otherwise it is worse. Beam search helps LTF considerably, such that it even catches up

	DM		**PAS**		**PSD**		**EDS**	**AMR 15**	**AMR 17**
	id	ood	id	ood	id	ood	test	test	test
LTL	88.8±0.7	82.3±0.5	87.9±1.1	82.2±0.2	65.0±1.6	64.6±0.6	82.0±0.9	45.4±1.8	51.2±2.3
LTF	79.5±1.7	73.3±0.2	84.2±1.0	77.0±1.2	60.1±1.1	60.4±0.7	70.1±0.6	36.7±1.0	41.2±1.1
LTL	90.5±0.2	87.3±0.2	88.6±1.3	85.5±1.1	73.5±0.9	76.3±1.1	84.4±0.7	50.8±1.2	57.8±1.3
LTF	84.6±0.1	80.2±0.8	87.3±0.6	82.6±0.6	68.5±1.5	70.4±0.8	72.9±0.3	41.2±0.3	46.9±2.1

Table 5.5: Proportion of well-typed AM dependency on test data when well-typedness is not enforced during greedy parsing. Models using BERT are below the line. Reported are means and standard deviations over the four runs.

with LTL in some cases. The positive effect of using beam search is more pronounced than for LTL: LTF improves by 2.6 points F-score on average whereas LTL only improves by 0.5 points F-score.

BERT When we use BERT embeddings, performance is generally much better and differences are a bit smaller. LTL with beam search is still overall the best parsing method, and only outperformed on DM out-of-domain and PSD in-domain; the differences are tiny however and might be just due to the random initializations. The improvement is largest again for AMR 17, where LTL with beam search performs up to 0.7 points F-score better than L'19. LTF performs worse than LTL to a similar degree as with GloVe embeddings.

Type constraints The rows in Table 5.3 indicated with "-types" denote the performance we achieve when we parse the test sets with the same models but without the type constraints, and completely rely on what the model has learned about well-typedness from the training data. We see that accuracy drops enormously, with the most extreme case being AMR. If a predicted AM dependency tree is not well-typed, we cannot evaluate it to a graph. In order to compute F-scores anyway, we return a graph without edges (DM, PAS, PSD) or a graph with a single dummy node (EDS, AMR). That is, a low number of well-typed trees among the predictions will drag the overall F-score considerably down. Table 5.5 shows the exact numbers of predicted AM dependency trees that are well-typed. Well-typedness seems to correlate with the F-scores the models achieve when the type constraints are enforced. For instance, this also means that well-typedness decreases if we leave the training domain or if we use LTF instead of LTL.

Comparison by sentence length Table 5.3 shows that the largest improvement of LTL over the existing parser is in AMR 2017. Here, we focus on the case where BERT is used, and want to see if L'19 and LTL perform systematically different by sentence length. We compare one run of LTL (beam size 3) to one run of L'19 + CharCNN (both using BERT) which achieve 76.9 and 76.2 points overall F-score on the test set, respectively. The runs were chosen based on how close they are to the average F-score on the test set.

Figure 5.1 shows average precision, recall and F-score by sentence length, for sentences with length between 3 and 60. It shows that LTL has better precision on average, irrespective of the sentence length. Recall is slightly better or comparable for lengths up 30 and then becomes slightly worse than L'19.

Looking at precision and recall overall, this translates to a higher precision for LTL (80.4 vs 78.5) but lower recall (73.6 vs 74.0) than L'19.

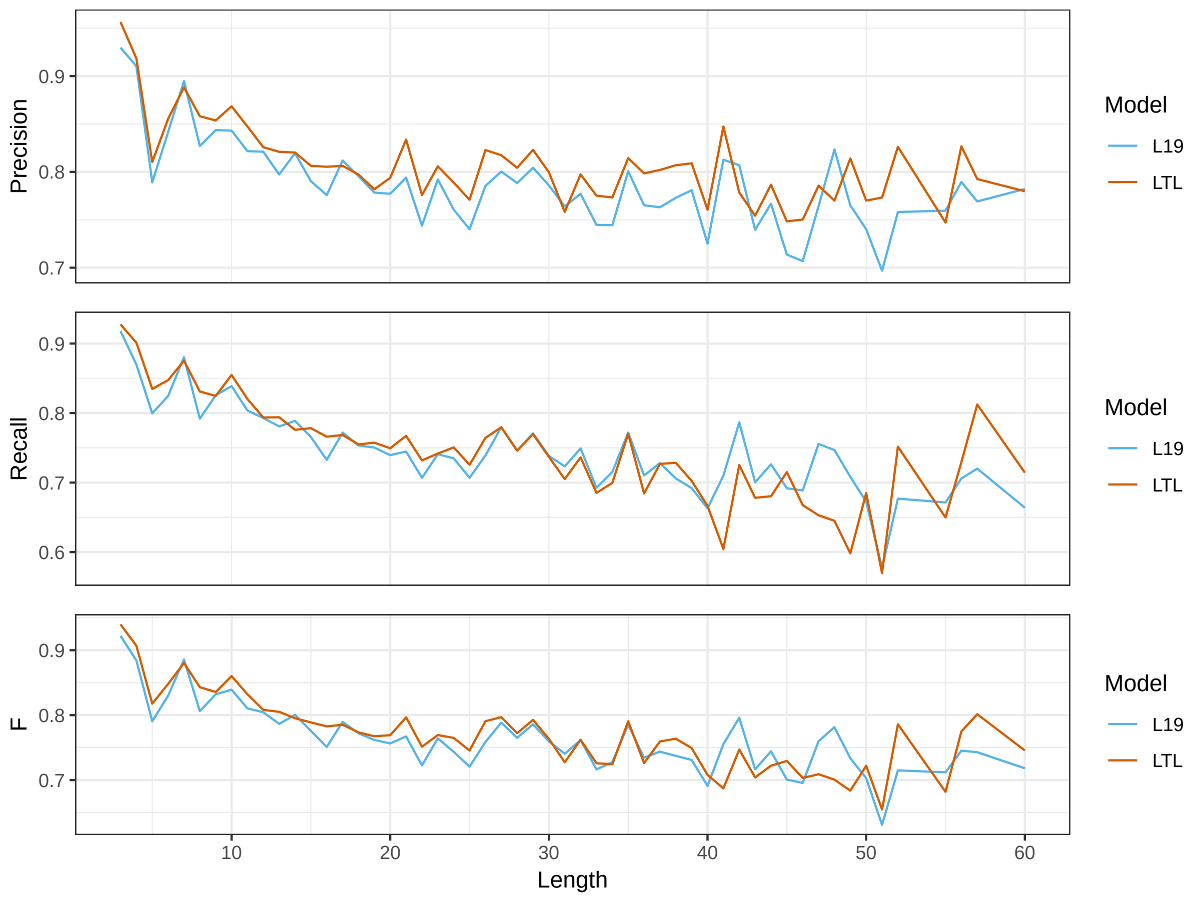


Figure 5.1: Averages of precision, recall, and F-score by sentence length for LTL (beam size 3) and L'19 +CharCNN on AMR 2017. All models use BERT.

Comparison of AM dependency trees If we consider the part of the development set where we have AM dependency trees, we can make a more in depth analysis, comparing different properties of the predicted AM dependency trees to the gold AM dependency trees. This is shown in Table 5.6. *Content precision* and *recall* measure how well the parsers can discover which tokens are part of the AM dependency tree, and which are not. The other metrics are computed such that they are robust regarding content precision and recall: *Constants* refers to the proportion of correctly predicted (delexicalized) graph constants given that both the prediction and the gold standard assign it *some* graph constant. The other metrics are computed analogously. A token is *attached* correctly if the predicted head is the same as the head in the gold AM dependency tree.

Models trained with LTL usually make the best predictions for graph constants, edge identities and edge labels. However, they have a lower content recall than both LTF and L'19, which hurts overall accuracy.

In this analysis, LTF performs quite well, often between LTL and L'19 or closely behind but it suffers from consistently lower content precision, which is up to 10 points lower.

Comparison to state-of-the-art Table 5.4 compares the accuracy of the best performing model (LTL, beam size 3) to current state-of-the-art on the different graphbanks. When using GloVe embeddings, LTL performs worse than the state-of-the-art. Using BERT lets the difference shrink considerably; in that case LTL achieves strong performance but is still often more than 0.5 points F-score worse than the best performing model.

The best performing model on every graphbank is a model specifically designed for

	DM			**PAS**			**PSD**			**EDS**			**AMR 15**			**AMR 17**		
	LTL	LTF	L'19	LTL	LTF	L'19	LTL	LTF	L'19	LTL	LTF	L'19	LTL	LTF	L'19	LTL	LTF	L'19
Constants	**96.4**	96.3	95.8	**95.1**	95.0	94.4	82.0	81.3	**82.2**	**92.6**	92.1	91.9	**70.6**	68.4	66.6	**71.2**	69.0	67.2
Lex. type	**96.6**	96.4	95.9	**95.3**	95.1	94.6	**87.8**	87.2	87.6	**95.1**	94.6	94.7	**77.4**	75.3	74.2	**78.8**	76.9	75.6
Lex. label	**98.6**	98.6	98.1	**100.0**	**100.0**	**100.0**	97.5	97.5	**97.9**	**97.0**	96.9	94.7	**90.7**	89.7	89.8	**91.1**	90.1	90.7
UA	**95.8**	95.6	95.2	**95.9**	95.6	93.8	**95.3**	94.3	94.6	**94.9**	94.3	93.9	**81.0**	78.0	75.6	**81.2**	79.0	76.4
LA	**95.1**	94.7	94.2	**95.2**	94.8	92.8	**90.0**	88.8	89.3	**93.6**	92.3	92.4	**71.7**	68.3	65.4	**72.8**	69.9	66.6
Content P	**99.6**	98.3	99.5	**100.0**	99.8	**100.0**	99.6	94.2	**99.7**	**99.2**	96.7	98.8	**95.7**	85.6	94.9	**95.7**	87.0	94.9
Content R	99.2	99.4	**99.5**	99.7	99.8	**100.0**	99.5	**99.7**	**99.7**	98.8	99.1	**99.6**	90.7	**93.8**	93.3	91.7	**94.1**	93.4
Content F	99.4	98.8	**99.5**	99.8	99.8	**100.0**	99.5	96.9	**99.7**	99.0	97.9	**99.2**	93.1	89.5	**94.1**	93.7	90.4	**94.1**

Table 5.6: Comparison of greedy LTL, greedy LTF and L'19,fixed tree+CharCNN predictions on decomposable development sets. Reported are means over all runs. All models use BERT. UA is unlabeled attachment, LA is labeled attachment.

a relatively narrow family of graphbanks. For example, models designed for DM, PAS and PSD exploit the fact that the nodes in the semantic graph are exactly the tokens in the sentence. Chen et al. (2018) use information about the syntax tree from which EDS is derived at training time. Lyu and Titov (2018) and Cai and Lam (2020) both use elaborate learning mechanisms to identify how AMR graphs align to their respective sentences and train end-to-end neural models.

In comparison to the only other general-purpose semantic parser, Zhang et al. (2019), we see that LTL is practically on par with it on AMR but outperforms it on DM by a margin of at least 1.7 points F-score.

5.4 Run time

The other important question that we want to answer is: how fast are the transition systems in practice. We measure how much time passes between feeding the sentence into the neural network until the AM dependency tree is fully computed. The neural network (including BERT) runs on the GPU; parsing happens on a single CPU core unless noted otherwise. For all GPU computations, we measure its contribution to the run time of a single sentence as the computation time of the batch divided by the batch size. All parsing experiments were performed on the same hardware (see Section 5.2). Sentences that were skipped due to timeout (again, see Section 5.2) are *not* counted towards the overall parsing time.

Table 5.7 shows the parsing speed as tokens per second on the test sets, for DM, PAS and PSD, this is the in-domain test set; parsing speed on the out-of-domain test sets is comparable. Note that the parsing speed on AMR is not directly comparable to the other ones because it is evaluated on a different test set (containing two sentences with more than 150 tokens). We see that the projective parser is consistently extremely slow, the fixed tree parser is sufficiently fast for some graphbanks like DM and EDS but also extremely slow for AMR. The transition systems achieve high parsing speed across the board but the implementation of LTL is faster than that of LTF. Beam search degrades parsing speed but it is still up to two orders of magnitude faster than the projective parser on AMR.

The highest performance can be achieved with LTL and greedy parsing on the GPU, which is not only a fast parser by comparison with existing AM dependency parsers but also in comparison with the current state-of-the-art parser for AMR (Cai and Lam, 2020). A decisive factor for how fast parsing is on the GPU is the size of the type lexicon Ω. Recall that the parsing algorithm for the GPU computes an array with size quadratic in Ω (see Section 4.4.2). The size of Ω depends on the graphbank (see Table 5.2) and is largest for the AMR graphbanks and PSD, which are also the graphbanks where parsing on the

	DM	PAS	PSD	EDS	AMR 15	AMR 17
Cai and Lam (2020), GPU, greedy	-	-	-	-		762±67
Cai and Lam (2020), GPU, beam=8						455±26
L'19, projective	3	2	4	4	<2	<2
L'19, fixed tree	710	97	265	542	<4	<3
LTL, greedy	1,094±13	913±91	1,126±32	968±61	879±98	962±46
LTL, beam=3	241±2	203±12	231±5	217±4	217±14	205±16
LTL, GPU, greedy	**4,750**±84	**4,570**±0	**2,742**±95	**4,443**±118	**1,977**±40	**2,116**±111
LTF, greedy	852±89	791±85	688±9	673±43	563±34	424±181
LTF, beam=3	145±6	123±8	96±0	108±2	100±3	76±27

Table 5.7: Avg. parsing speed in tokens/s on test sets. < indicates where parsing was interrupted due to timeout. All models use BERT.

	DM	PAS	PSD	EDS	AMR 15	AMR 17
LTL, greedy	1,180±152	1,128±11	1,288±39	1,154±26	1,121±110	1,162±29
LTL, beam=3	257±13	229±5	243±4	224±9	234±14	222±13
LTL, GPU, greedy	**10,266**±182	**10,271**±307	**4,201**±171	**9,188**±528	**3,647**±222	**3,413**±154
LTF, greedy	957±67	908±111	755±12	752±35	672±34	578±83
LTF, beam=3	153±2	126±10	97±2	113±2	104±5	91±4

Table 5.8: Avg. parsing speed in tokens/s on (in-domain) test sets without BERT.

GPU is the slowest.

Table 5.8 shows a comparison of parsing speed on the test sets for the case where we use GloVe embeddings instead of BERT. Since BERT takes considerable time to compute, parsing is faster but only slightly for models that perform inference on the CPU, indicating that this is the bottleneck. On the GPU, parsing speed increases much more, reaching up to 10,000 tokens per second.

Figure 5.2 compares the run times of the fixed tree decoder, the projective parser and LTL (greedy, GPU) as a function of sentence length on a selection of the graphbanks. Note the that the parsing time is on a log scale. As expected, the run time of the projective parser increases quite quickly with longer sentences because its time complexity is $O(n^5)$. The fixed-tree decoder is generally faster than the projective parser but there are outliers where the tree has a high out-degree. Since the fixed-tree decoder has exponential run time in the maximum out-degree, parsing can take more than 1,000 seconds for such a sentence. LTL on the GPU is extremely fast and as expected, run time increases relatively mildly for longer sentences although the exact relationship between run time of a single sentence and its length is hard to determine because of the large batches.

5.5 Discussion

The evaluation shows that the LTL transition system can give very strong performance in practice, which is on par with or even better than existing AM dependency parsers and also achieves strong performance in comparison with state-of-the-art models for the different graphbanks. Since AM dependency parsing is a flexible way of semantic parsing that can be applied to many graphbanks, it is particularly interesting to compare it to other general purpose semantic parsers like that of Zhang et al. (2019). This comparison is relatively favorable because LTL outperforms Zhang et al. (2019) on DM and is comparable with them on AMR.

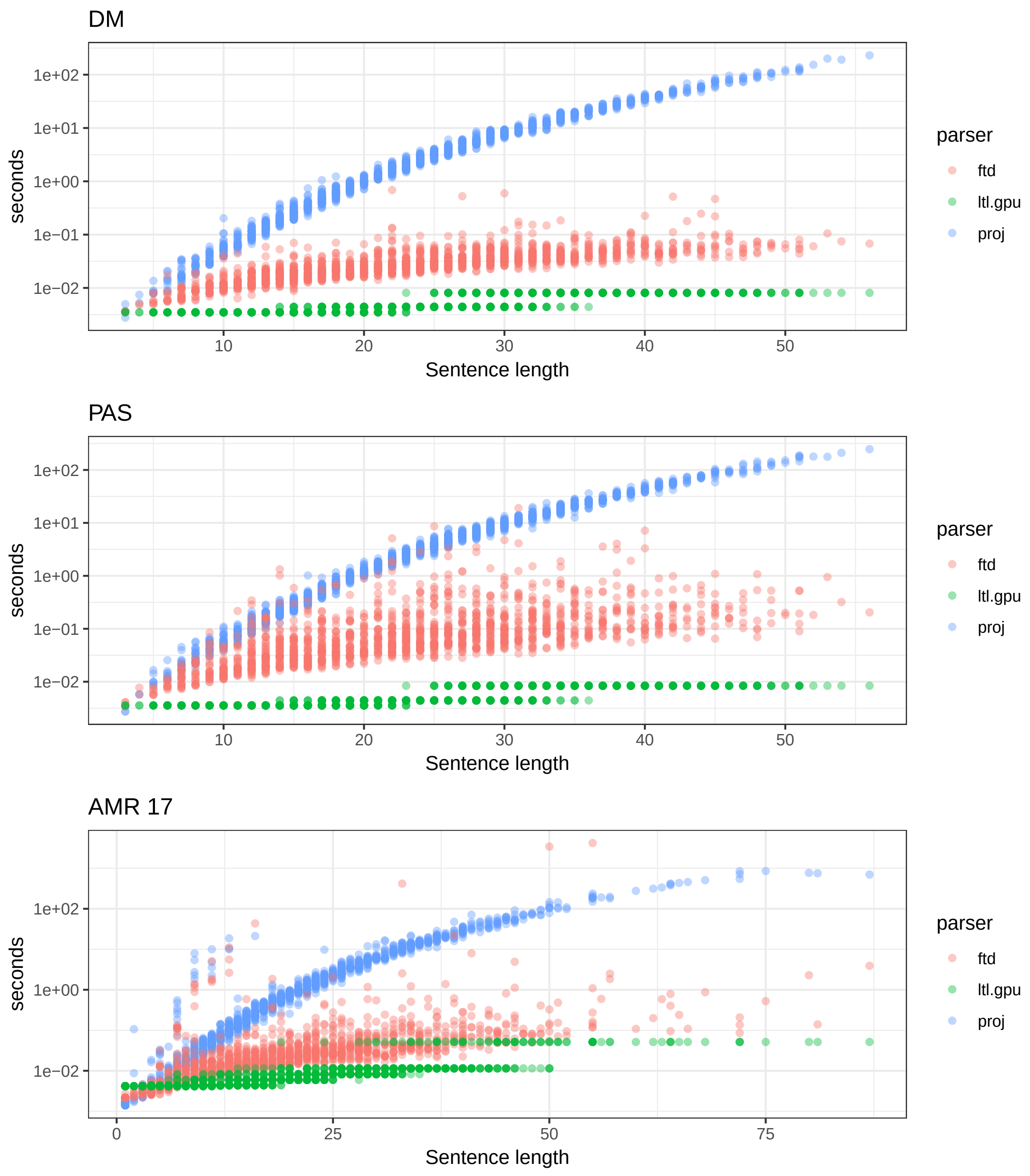


Figure 5.2: Parsing time on (in-domain) test sets as function of sentence length, up to sentence length 100. All models use BERT.

In terms of run time, we can see an improvement by a factor of up to 700 compared to the fixed-tree parser and up to 2200 compared to the projective parser of Groschwitz et al. (2018) when using BERT and parsing on the GPU. LTL is not only fast on a relative scale but also fast enough to be used in practice to parse large corpora.

Further speed improvements can be achieved by parsing without type constraints first, then only those sentences where the AM dependency tree is not well-typed have to be re-parsed with the type constraints. A way to increase parsing speed particularly on the GPU is to prune the set of possible types Ω at parsing time. The frequency of the types follows a Zipfian distribution, so pruning ones that are rarely used could increase speed at little cost in accuracy.

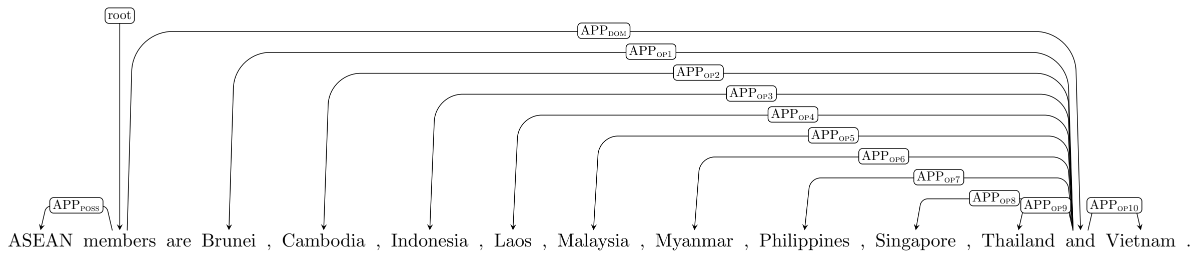

root
APP_DOM
APP_OP1
APP_OP2
APP_OP3
APP_OP4
APP_OP5
APP_OP6
APP_OP7
APP_OP8
APP_OP9
APP_OP10
APP_POSS
ASEAN members are Brunei , Cambodia , Indonesia , Laos , Malaysia , Myanmar , Philippines , Singapore , Thailand and Vietnam .

Figure 5.3: AM dependency tree (without constant symbols) for a large coordination in AMR.

The LTF transition system is fast as well but not as accurate as LTL and needs beam search to achieve good performance. It also struggles with content precision. We think that this is the result of error propagation. When LTF predicts a wrong graph constant, it has to draw outgoing edges that fit to this choice. As a result, LTF is sometimes forced to assign graph constants to tokens because of previous errors, not because the model assigns high probability to attaching that token. This then results in low content precision. Beam search can help here particularly because it allows the model to revise a choice of graph constant if that leads to bad outgoing edges later.

An example that can illustrate this problem of LTF is coordination with many conjuncts like in Figure 5.3. Here, a model that predicts LTF transitions has to guess the constant of *and* correctly, in particular how many conjuncts *and* coordinates. If the guess is off, the transition system forces us to either omit App edges or to draw more of them. Such a case is easier for LTL where we can predict one App edge after the other and the transition system keeps track of the number of conjuncts attached already and then chooses a fitting graph constant as soon as the model decides to stop adding edges.

While LTF struggles with content precision, LTL struggles with content recall – although to a lesser a degree. We think this is caused by the pure top-down perspective. The parser ignores a specific token if it is never deemed to be a suitable child for the token on top of the stack. However, this misses the bottom-up perspective that some tokens, in particular content words, are extremely likely to have *some* incoming edge. Taking into account bottom-up information is thus a promising research direction to improve the parser.

One very plausible reason for the models not learning to predict only well-typed AM dependency trees is that they do not condition directly on features of the generated tree that are important for well-typedness. In particular, the models do not receive the labels of outgoing edges as input, and they do not condition on the lexical types that have been assigned already. Changing that could increase the proportion of well-typed trees but comes at a risk as well: the more we condition on parts of the tree that have been built already, the stronger becomes the exposure bias where the model achieves sub-optimal performance because it overestimates the quality of what is has generated so far. Dynamic oracles (Goldberg and Nivre, 2013) are a standard method in dependency parsing that could help here although it might be difficult to develop them for AM dependency parsing.

One drawback of the models used here in comparison to the existing parser of L'19 is that of long training times: it takes up to 24 hours to train a model to convergence, whereas training an L'19 model only takes up to 11 hours. Future work can look into how efficiency *at training time* can be achieved with transition-based models by thoroughly investigating how to fully saturate the GPU during training, and experimenting with different hyperparameters and other model architectures.

Chapter 6

Conclusion

This thesis presented two transition systems for AM dependency parsing, which enable us to perform fast broad-coverage semantic parsing for a variety of graphbanks. Combining these transition systems with neural dependency parsing models in the style of Ma et al. (2018) yields accuracies that are competitive with state-of-the-art models and on par with or better than existing AM dependency parsers.

Parsing with the transition systems is fast both asymptotically and in practice, achieving a speed-up of several orders of magnitude compared to existing AM dependency parsers. The fastest model parses more than 10,000 tokens per second on the GPU.

The transition systems also exhibit a number of desirable theoretical properties. In particular, they are sound, complete and – under mild conditions – free of dead ends, i.e. every configuration can be completed to a goal configuration without needing to backtrack.

Future work can make use of the fact that these parsers are fast and experiment with reinforcement learning, e.g. to maximize F-score directly instead of maximizing the log-likelihood. The top-down approach also makes it easy to phrase generative models of well-typed AM dependency trees, which could be used for semi-supervised or even unsupervised AM dependency parsing. The LTL inference algorithm on GPUs could be a starting point for continuous relaxations, which could enable us to perform variational inference efficiently.

Finally, the transition systems can be adapted for other semantic algebras, and can perhaps be also applied to problems such as TAG parsing.

Bibliography

Omri Abend and Ari Rappoport. 2013. Universal conceptual cognitive annotation (UCCA). In *Proceedings of the ACL*. https://www.aclweb.org/anthology/P13-1023.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Proceedings of the ACL*. https://doi.org/10.18653/v1/P16-1231.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Deng Cai and Wai Lam. 2020. AMR parsing via graph-sequence iterative inference. In *Proceedings of the ACL*. https://www.aclweb.org/anthology/2020.acl-main.119.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the ACL*.

Yufei Chen, Weiwei Sun, and Xiaojun Wan. 2018. Accurate SHRG-based semantic parsing. In *Proceedings of the ACL*. https://www.aclweb.org/anthology/P18-1038.

Bruno Courcelle and Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic, a Language Theoretic Approach*. Cambridge University Press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the NAACL-HLT*. https://www.aclweb.org/anthology/N19-1423.

Lucia Donatelli, Meaghan Fowlie, Jonas Groschwitz, Alexander Koller, Matthias Lindemann, Mario Mina, and Pia Weißenhorn. 2019. Saarland at MRP 2019: Compositional parsing across all graphbanks. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. https://doi.org/10.18653/v1/K19-2006.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *ICLR*.

Timothy Dozat and Christopher D. Manning. 2018. Simpler but more accurate semantic dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*. http://aclweb.org/anthology/P18-2077.

Rebecca Dridan and Stephan Oepen. 2011. Parser evaluation using elementary dependency matching. In *Proceedings of the 12th International Conference on Parsing Technologies, IWPT*. https://www.aclweb.org/anthology/W11-2927/.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2020. Transition-based semantic dependency parsing with pointer networks. In *Proceedings of the ACL*. https://www.aclweb.org/anthology/2020.acl-main.629.

Yarin Gal and Zoubin Ghahramani. 2016. A theoretically grounded application of dropout in recurrent neural networks. In *NIPS*. http://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.

Matthew Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson H S Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. A deep semantic natural language processing platform.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Trans. Assoc. Comput. Linguistics* 1:403–414. https://tacl2013.cs.columbia.edu/ojs/index.php/tacl/article/view/145.

Jonas Groschwitz. 2019. *Methods for taking semantic graphs apart and putting them back together again*. Ph.D. thesis, Macquarie University and Saarland University. https://doi.org/10.22028/D291-30370.

Jonas Groschwitz, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2017. A constrained graph algebra for semantic parsing with amrs. In *IWCS 2017-12th International Conference on Computational Semantics*.

Jonas Groschwitz, Matthias Lindemann, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2018. AMR Dependency Parsing with a Typed Semantic Algebra. In *Proceedings of the ACL*.

Han He and Jinho Choi. 2020. Establishing strong baselines for the new decade: Sequence tagging, syntactic and semantic parsing with bert. In *The Thirty-Third International Flairs Conference*.

Jungo Kasai, Bob Frank, Tom McCoy, Owen Rambow, and Alexis Nasr. 2017. TAG parsing with neural networks and vector representations of supertags. In *Proceedings of EMNLP*. https://doi.org/10.18653/v1/D17-1180.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*. http://arxiv.org/abs/1412.6980.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics* 4:313–327.

Alexander Koller and Marco Kuhlmann. 2011. A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*. https://www.aclweb.org/anthology/W11-2902.

Mike Lewis and Mark Steedman. 2014. Improved CCG parsing with semi-supervised supertagging. *Transactions of the Association for Computational Linguistics* https://www.aclweb.org/anthology/Q14-1026.

Matthias Lindemann. 2018. *AMR Parsing as Typed Dependency Parsing with a Graph Algebra*. Bachelor's thesis, Saarland University.

Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. 2019. Compositional semantic parsing across graphbanks. In *Proceedings of the ACL*. https://www.aclweb.org/anthology/P19-1450.

Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. 2020. Fast semantic parsing with well-typedness guarantees. Under review.

Chunchuan Lyu and Ivan Titov. 2018. AMR Parsing as Graph Prediction with Latent Alignment. In *Proceedings of the ACL*.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. 2018. Stack-pointer networks for dependency parsing. In *Proceedings of the ACL*. https://doi.org/10.18653/v1/P18-1130.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of the ACL*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics* 19(2):313–330.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France. https://www.aclweb.org/anthology/W03-3017.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In Nicoletta Calzolari, Khalid Choukri, Aldo Gangemi, Bente Maegaard, Joseph Mariani, Jan Odijk, and Daniel Tapias, editors, *Proceedings of LREC*. http://www.lrec-conf.org/proceedings/lrec2006/summaries/162.html.

Stephan Oepen, Omri Abend, Jan Hajic, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdenka Uresova. 2019. MRP 2019: Cross-framework meaning representation parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 CoNLL*. https://doi.org/10.18653/v1/K19-2001.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajič, and Zdeňka Urešová. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. http://aclweb.org/anthology/S15-2153.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based MRS banking. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*.

Adam Paszke, Sam Gross, Francisco Massa, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NEURIPS*. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Antoine Venant and Alexander Koller. 2019. Semantic expressive capacity with bounded memory. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. https://doi.org/10.18653/v1/P19-1008.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS*. http://papers.nips.cc/paper/5866-pointer-networks.

Xinyu Wang, Jingxian Huang, and Kewei Tu. 2019. Second-order semantic dependency parsing with end-to-end neural networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. https://doi.org/10.18653/v1/P19-1454.

Dani Yogatama, Phil Blunsom, Chris Dyer, Edward Grefenstette, and Wang Ling. 2017. Learning to compose words into sentences with reinforcement learning. *ICLR* .

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019. Broad-coverage semantic parsing as transduction. In *Proceedings of the EMNLP-IJCNLP*. https://doi.org/10.18653/v1/D19-1392.

Appendix A

Hyperparameters

We set the hyperparameters manually without extensive hyperparameter search, following mostly Ma et al. (2018) where applicable and otherwise following Lindemann et al. (2019).

Following Groschwitz et al. (2018); Lindemann et al. (2019), we split the prediction of a graph constant into predicting a delexicalized graph constant and a lexical label.

We train all LTL and LTF models for 100 epochs with the Adam optimizer (Kingma and Ba, 2015) using a batch size of 64. We follow Ma et al. (2018) in setting $\beta_1, \beta_2 = 0.9$ and the initial learning rate to 0.001. We do not perform weight decay or gradient clipping. We parse the full development set after each epoch and choose the model with the highest F-score (for EDS: measured in Smatch, not in EDM).

Training an LTL or LTF model with BERT took at most 24 hours, and about 10 hours for AMR 15. Training with GloVe usually saves two or three hours.

In experiments with GloVe, we use the vectors of dimensionality 200 (6B.200d), and further train the embeddings. We use BERT in the variant large-uncased, exactly as Lindemann et al. (2019).

We perform variational dropout (Gal and Ghahramani, 2016) on $\mathbf{x}'$ with $p = 0.33$, as well as on $\mathbf{s}$ and $\mathbf{s}'$. The hyperparameters are listed in Table A.1. *Other MLPs* refers to the MLPs we use for predicting delexicalized constants, term types (only needed for LTF) and lexical labels. In case of using BERT embeddings, we thought that 2 layers in the LSTM for $\mathbf{s}'$ might be enough since we use it to predict graph constant and thus it has to consider limited context compared to $\mathbf{s}$, which is used for predicting dependency edges. Lindemann et al. (2019) also reduce the number of LSTM layers when using BERT. We did not check the impact on performance of this.

All LSTMs:	
LSTM hidden size (per direction)	512
LSTM layer dropout	0.33
LSTM recurrent dropout	0.33
LSTM layers used for $\mathbf{s}$	3
LSTM layers used for $\mathbf{s}'$	3 / 2
Decoder LSTM layers	1
MLPs before bilinear scoring	
Layers	1
Hidden units	512
Activation	elu
Dropout	0.33
Edge label model	
Layers	1
Hidden units	256
Activation	tanh
Dropout	0.33

Other MLPs	
Layers	1
Hidden units	1024
Activation	tanh
Dropout	0.4
Character CNN	
Filters	50
Window size	3
Embeddings	
POS	32
Characters	100
NE embedding	16

Table A.1: Hyperparameters of neural models. 3 / 2 means that the value is 3 when using GloVe word embeddings and 2 when using BERT.