Fast semantic parsing with well-typedness guarantees

Matthias Lindemann and **Jonas Groschwitz** and **Alexander Koller**
Department of Language Science and Technology
Saarland University
{mlinde|jonasg|koller}@coli.uni-saarland.de

Abstract

AM dependency parsing is a linguistically principled method for neural semantic parsing with high accuracy across multiple graphbanks. It relies on a type system that models semantic valency but makes existing parsers slow. We describe an A* parser and a transition-based parser for AM dependency parsing which guarantee well-typedness and improve parsing speed by up to 3 orders of magnitude, while maintaining or improving accuracy.

1 Introduction

Over the past few years, the accuracy of neural semantic parsers which parse English sentences into graph-based semantic representations has increased substantially (Dozat and Manning, 2018; Zhang et al., 2019; He and Choi, 2020; Cai and Lam, 2020). Most of these parsers use a neural model which can freely predict node labels and edges, and most of them are tailored to a specific type of graphbank.

Among the high-accuracy semantic parsers, the *AM dependency parser* of Groschwitz et al. (2018) stands out in that it implements the Principle of Compositionality from theoretical semantics in a neural framework. By parsing into AM dependency trees, which represent the compositional structure of the sentence and evaluate deterministically into graphs, this parser can abstract away surface details of the individual graphbanks. It was the first semantic parser which worked well across multiple graphbanks, and set new states of the art on several of them (Lindemann et al., 2019).

However, the commitment to linguistic principles comes at a cost: the AM dependency parser is slow. A key part of the parser is that AM dependency trees must be *well-typed* according to a type system which ensures that the semantic valency of each word is respected. Existing algorithms compute all items along a parsing schema that encodes the type constraints; they parse e.g. the AMRBank at less than three tokens per second.

In this paper, we present two fast and accurate parsing algorithms for AM dependency trees. We first present an A* parser which searches through the parsing schema of Groschwitz et al.'s "projective parser" efficiently (§4). We extend the supertag-factored heuristic of Lewis and Steedman's (2014) A* parser for CCG with a heuristic for dependency edge scores. This parser achieves a speed of up to 2200 tokens/s on semantic dependency parsing (Oepen et al., 2015), at no loss in accuracy. On AMR corpora (Banarescu et al., 2013), it achieves a speedup of 10x over previous work, but still does not exceed 30 tokens/second.

We therefore develop an entirely new transition-based parser for AM dependency trees, inspired by the stack-pointer parser of Ma et al. (2018) for syntactic dependency parsing (§5). The key challenge here is to adhere to complex symbolic constraints – the AM algebra's type system – without running into dead ends. This is hard for a greedy transition system and in other settings requires expensive workarounds, such as backtracking. We ensure that our parser avoids dead ends altogether. We define two variants of the transition-based parser, which choose types for words either before predicting the outgoing edges or after, and introduce a neural model for predicting transitions. In this way, we guarantee well-typedness with $O(n^2)$ parsing complexity, achieve a speed of several thousand tokens per second across all graphbanks, and even improve the parsing accuracy over previous AM dependency parsers by up to 1.6 points F-score.

2 Related work

In *transition-based parsing*, a dependency tree is built step by step using nondeterministic transitions. A classifier is trained to choose transitions deter-

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pages 3929–3951,
November 16–20, 2020. ©2020 Association for Computational Linguistics

ministically (Nivre, 2008; Kiperwasser and Goldberg, 2016). Transition-based parsing has also been used for constituency parsing (Dyer et al., 2016) and graph parsing (Damonte et al., 2017). We build most directly upon the top-down parser of Ma et al. (2018). Unlike most other transition-based parsers, our parser implements hard symbolic constraints in order to enforce well-typedness. Such constraints can lead transition systems into dead ends, requiring the parser to backtrack (Ytrestøl, 2011) or return partial analyses (Zhang and Clark, 2011). Our transition system carefully avoids dead ends. Shi and Lee (2018) take hard valency constraints into account in chart-based syntactic dependency parsing, avoiding dead ends by relaxing the constraints slightly in practice.

A* parsing is a method for speeding up agenda-based chart parsers, which takes items off the agenda based on a heuristic estimate of completion cost. A* parsing has been used successfully for PCFGs (Klein and Manning, 2003), TAG (Bladier et al., 2019), and other grammar formalisms. Our work is based most closely on the CCG A* parser of Lewis and Steedman (2014).

Most approaches that produce semantic graphs (see Koller et al. (2019) for an overview) model distributions over graphs directly (Dozat and Manning, 2018; Zhang et al., 2019; He and Choi, 2020; Cai and Lam, 2020), while others make use of derivation trees that compositionally evaluate to graphs (Groschwitz et al., 2018; Chen et al., 2018; Fancellu et al., 2019; Lindemann et al., 2019). AM dependency parsing belongs to the latter category.

3 Background

We begin by sketching the AM dependency parser of Groschwitz et al. (2018).

3.1 AM dependency trees

Groschwitz et al. (2018) use *AM dependency trees* to represent the compositional structure of a semantic graph. Each token is assigned a *graph constant* representing its lexical meaning; dependency labels correspond to operations of the *Apply-Modify (AM) algebra* (Groschwitz et al., 2017; Groschwitz, 2019), which combine graphs into bigger ones.

Fig. 2 illustrates how an AM dependency tree (a) evaluates to a graph (b), based on the graph constants in Fig. 1. Each graph constant is an *as-graph*, which means it has special node markers called *sources*, drawn in red, as well as a *root* marked in bold. These markers are used to combine graphs with the algebra's operations. For instance, the MOD_M operation in Fig. 2a combines the head G_sleep with its modifier G_soundly by plugging the root of G_sleep into the M-source of G_soundly, see (c). That is, G_soundly has now modified G_sleep and (c) is our graph for *sleep soundly*. The other operation of the AM algebra, APP, models argument application. For example, the APP_O operation in Fig. 2a plugs the root of (c) into the O source of G_want. Note that because G_want and (c) both have an S-source, APP_O merges these nodes, see (d). The APP_S operation then fills this S-source with G_writer, attaching the graph with its root, to obtain the final graph in (b).[1]

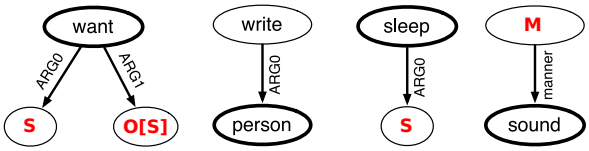

Figure 1: Elementary as-graphs G_want, G_writer, G_sleep, and G_sound.

Types. The [S] annotation at the O-source of G_want is a *request* as to what the *type* of the O argument of G_want should be. The type of an as-graph is the set of its sources with their request annotations, so the request [S] means that the source set of the argument must be {S}. Because this is true for (c), the AM dependency tree is *well-typed*; otherwise the tree could not be evaluated to a graph. Thus, the graph constants lexically specify the semantic valency of each word as well as reentrancies due to e.g. control, like in this example.

If an as-graph has no sources, we say it has the *empty type* []; if a source in a graph printed here has no annotation, it is assumed to have the empty request (i.e. its argument must have no sources). We write τ_G for the type of an as-graph G, and $req_\alpha(\tau)$ for the request at source α of type τ. For example, $req_\text{O}(\tau_{G_\text{want}}) = [\text{S}]$ and $req_\text{S}(\tau_{G_\text{want}}) = []$. If an AM dependency (sub-)tree evaluates to a graph G, we call τ_G its *term type*. For example, the sub-tree in Fig. 2a rooted in *sleep* has term type [S], since it evaluates to (c).

Below, we will build AM dependency trees by adding the outgoing edges of a node one by one;

[1] When evaluating an AM dependency tree, the AM algebra restricts operation orders to ensure that every AM dependency tree evaluates to a unique graph. For instance, in Fig. 2, the APP_O edge out of "wants" is always tacitly evaluated before the APP_S edge. For details on this, we refer to Groschwitz et al. (2018) and Groschwitz (2019).

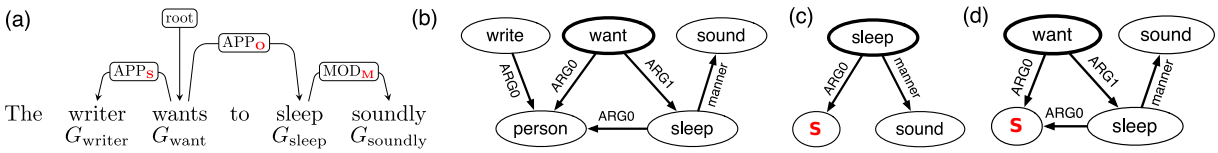

Figure 2: (a) An AM dependency tree with its evaluation result (b), along with two partial results (c) and (d).

we track types there with the following notation. If τ_1 and τ_2 are the term types of AM dependency trees t_1, t_2 and ℓ is an operation of the AM algebra, we write $\ell(\tau_1, \tau_2)$ for the term type of the tree constructed by adding t_2 as an ℓ-child to t_1, i.e. by adding an ℓ-edge from the root of t_1 to the root of t_2 (if that tree is well-typed). Intuitively, one can see this as combining a graph of type τ_1 (the head) with an argument or modifier of type τ_2 using operation ℓ; the result then has type $\ell(\tau_1, \tau_2)$.

3.2 AM dependency parsing

Groschwitz et al. (2018) approach graph parsing as first predicting a well-typed AM dependency tree for a sentence $w_1, \ldots, w_n$ and then evaluating it deterministically to obtain the graph.

They train a *supertag and edge-factored model*, which predicts a *supertag cost* $c(G, i)$ for assigning a graph constant G to the token w_i, as well as an *edge cost* $c\left(i \xrightarrow{\ell} j\right)$ for each potential edge from word w_i to w_j with label ℓ. Tokens which are not part of the AM dependency tree, like *the* and *to* in Fig. 2a, are treated as if they were assigned the special graph constant $\bot$ and an incoming 'IGNORE' edge $0 \xrightarrow{\text{IGNORE}} i$, where 0 represents an artificial root. The root of the AM dependency tree (*wants* in the example) is modeled as having an incoming edge $0 \xrightarrow{\text{ROOT}} i$.

An algorithm for AM dependency parsing searches for the well-typed AM dependency tree which minimizes the sum of supertag and edge costs. Finding the lowest-cost well-typed AM dependency tree for a given sentence is NP-complete. Groschwitz et al. define two approximate parsing algorithms, the 'fixed tree decoder' that fixes an unlabeled dependency tree first , and the 'projective decoder'. Our A* parser is based on the projective decoder and we focus on it here.

Projective decoder. The projective decoder circumvents the NP-completeness by searching for the best *projective* well-typed AM dependency tree. It derives parsing items using the schema (Shieber et al., 1995) shown in Fig. 3[2].

[2]Originally only the fixed tree decoder used IGNORE and ROOT edge scores; we extend the projective decoder here for consistency.

$$\frac{s = c(G, i) \quad G \neq \bot}{([i, i+1], i, \tau_G) : s} \text{ Init}$$

$$\frac{([i,k], r, \tau) : s \quad s' = c(\bot, k) + c\left(0 \xrightarrow{\text{IGNORE}} k\right)}{([i, k+1], r, \tau) : s + s'} \text{ Skip-R}$$

$$\frac{([i,k], r, \tau) : s \quad s' = c(\bot, i-1) + c\left(0 \xrightarrow{\text{IGNORE}} i-1\right)}{([i-1, k], r, \tau) : s + s'} \text{ Skip-L}$$

$$\frac{([i,j], r_1, \tau_1) : s_1 \quad ([j,k], r_2, \tau_2) : s_2 \quad \tau = \ell(\tau_1, \tau_2) \text{ defined} \quad s = c\left(r_1 \xrightarrow{\ell} r_2\right)}{([i,k], r_1, \tau) : s_1 + s_2 + s} \text{ Arc-R } [\ell]$$

$$\frac{([i,j], r_1, \tau_1) : s_1 \quad ([j,k], r_2, \tau_2) : s_2 \quad \tau = \ell(\tau_2, \tau_1) \text{ defined} \quad s = c\left(r_2 \xrightarrow{\ell} r_1\right)}{([i,k], r_2, \tau) : s_1 + s_2 + s} \text{ Arc-L } [\ell]$$

$$\frac{([1, n+1], r, [\,]) : \quad s' = c\left(0 \xrightarrow{\text{ROOT}} r\right)}{([0, n+1], r, [\,]) : s + s'} \text{ Root}$$

Figure 3: Rules for the projective and A* decoder.

Each item encodes properties of a partial AM dependency tree and has the form $([i,k], r, \tau) : s$, where $[i,k] = \{j \mid i \leq j < k\}$ is the span of word indices covered by the item, r is the index of the head word, τ the type and s the cost. The Init rule assigns a supertag G to a word w_i. The Skip-R and Skip-L rules extend a span without changing the dependency derivation, effectively skipping a word by assigning it the $\bot$ supertag and drawing the corresponding 'IGNORE' edge. Finally the Arc-R and Arc-L rules, for an AM operation ℓ, combine two items covering adjacent spans by drawing an edge with label ℓ between their heads. Once the full chart is computed, i.e. all items are explored, a Viterbi algorithm yields the highest scoring well-typed AM dependency tree.

The projective decoder has an asymptotic runtime of $O(n^5)$ in the sentence length n.

Notation and terminology. Below, we assume that we obtained three fixed non-empty finite sets in training: a set Ω of types; a set C of graph constants (the *graph lexicon*) such that Ω is the set of types of the graphs in C; and a set L of operations, including ROOT and IGNORE. We write S for the set of sources occurring in C and assume

that for every source $s \in S$, $\text{APP}_s \in L$. We write $Dom(f)$ for the domain of a partial function f, i.e. the set of objects for which f is defined.

4 A* AM dependency parsing

While the AM dependency parser yields strong accuracies across multiple graphbanks, Groschwitz et al.'s algorithms are quite slow in practice. For instance, the projective parser needs several hours to parse each test set in §6, which seriously limits its practical applicability. In this section, we will speed the projective parser up through A* search.

4.1 Agenda-based A* parsing

Our A* parser maintains an *agenda* of parse items of the projective parser. The agenda is initialized with the items produced by the Init rule. Then we iterate over the agenda. In each step, we take the item I from the front of the agenda and apply the rules Skip-L and Skip-R to it. We also attempt to combine I with all previously discovered items, organized in a parse chart, using the Arc-L and Arc-R rules. All items thus generated are added to the agenda and the chart. Parsing ends once we either take a goal item $([0, n+1], r, [\,])$ from the agenda, or (unsucessfully) when the agenda becomes empty.

A* parsers derive their efficiency from their ability to order the items on the agenda effectively. They sort the agenda in ascending order of estimated cost $f(I) = c(I) + h(I)$, where c is the cost derived for the item I by the parsing rules in Fig. 3 and $h(I)$ is an *outside estimate*. The quantity $h(I)$ estimates the difference in cost between I and the lowest-cost well-typed AM dependency tree t that contains I. An outside heuristic is *admissible* if it is optimistic with respect to cost, i.e. $f(I) \leq c(t)$; in this case the parser is provably optimal, i.e. the first goal item which is dequeued from the agenda describes the lowest-cost parse tree. Tighter outside estimates lead to fewer items being taken off the agenda and thus to faster runtimes.

A first *trivial*, but admissible baseline lets $h(I) = 0$ for all items I. This ignores the outside part and orders items purely on their past cost. We could obtain a better outside heuristic by following Lewis and Steedman (2014) and summing up the cost of the lowest-cost supertag for each token outside of the item, i.e.

$$h([i,k], r, \tau) = \sum_{j \notin [i,k]} \min_G c(G, j).$$

This heuristic is admissible because each token will have *some* supertag selected (perhaps $\bot$) in a complete AM dependency tree, and its cost will be equal or higher than that of the best supertag.

4.2 Edge-based A* heuristics

Both of these outside heuristics ignore the fact that the cost of a tree consists not only of the cost for the supertags, but also of the cost for the edges.

We can obtain tighter heuristics by taking the edges into account. Observe first that the parse item determines the supertags and edges within its substring, and has designated one of the tokens as the root of the subtree it represents. For all tokens outside of the span of the item, the best parse tree will assign both a supertag to the token (potentially $\bot$) and an incoming edge (potentially with edge label ROOT or IGNORE). Thus, we obtain an admissible *edge-based heuristic* by adding the lowest-cost incoming edge for each outside token as follows:

$$h([i,k], r, \tau) = \sum_{j \notin [i,k]} \min_G c(G, j) + \min_{o \xrightarrow{\ell} j} c(o \xrightarrow{\ell} j)$$

Observe finally that the edge-based heuristic is still overly optimistic, in that it assumes that arbitrarily many nodes in the tree may have incoming ROOT edges (when it needs to be exactly one), and that the choice of IGNORE and $\bot$ are independent (when a node should have an incoming IGNORE edge if and only if its supertag is $\bot$). We can optimize it into the *ignore-aware outside heuristic* by restricting the $\min$ operations so they respect these constraints.

5 Transition-based parsing

As we will see in §6, the A* parser is very efficient on the DM, PAS, and PSD corpora but still slow on EDS and AMR.

Therefore, we develop a novel transition-based algorithm for AM dependency parsing. Inspired by the syntactic dependency parser of Ma et al. (2018), it builds the dependency tree top-down, starting at the root and recursively adding outgoing edges to nodes. However, for AM dependency parsing we face an additional challenge: we must assign a type to each node and ensure that the overall AM dependency tree is well-typed.

We will first introduce some notation (§5.1), then introduce three versions of our parsing schema (§5.2-§5.4), give theoretical guarantees (§5.5) and define the neural model (§5.6).

5.1 Apply sets

The transition-based parser chooses a graph constant G_i for each token w_i; we call its type, τ_{G_i}, the *lexical type* λ of w_i. As we add outgoing edges to i, each outgoing APP_α operation consumes the α source of the lexical type. To produce a well-typed AM dependency tree of term type τ, the sources of outgoing APP edges at i must correspond to exactly the *apply set* $\mathcal{A}(\lambda, \tau)$, which is defined as the set $O = \{o_1, \ldots, o_n\}$ of sources such that

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

for some types $\tau_1, \ldots, \tau_n$. That is, the apply set $\mathcal{A}(\lambda, \tau)$ is the set of sources we need to consume to turn λ into τ.

Note that there are pairs of types for which no such set of sources exists; e.g. the apply set $\mathcal{A}([\,], [s])$ is not defined. In that case, we say that $[s]$ is not *apply-reachable* from $[\,]$; the term type must always be apply-reachable from the lexical type in a well-typed tree.

5.2 Lexical type first (with dead ends)

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 the following:

- $\mathbb{E}(i)$: the labeled incoming edge of i, written $j \xrightarrow{\ell} i$, where j is the head and l the label;
- $\mathbb{T}(i)$: the set of possible term types at i;
- $\mathbb{A}(i)$: the sources of outgoing APP edges at i, i.e. which sources of the apply set we have covered;
- $\mathbb{G}(i)$: 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 $\mathbb{A}(i) = \mathcal{A}(\tau_G, \tau)$, i.e. $\mathbb{A}(i)$ must be the apply set for the lexical type τ_G and the term type τ. There must be at least one token that is not ignored.

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 Fig. 4.

INIT. An $\text{INIT}(i)$ transition is always the first transition and makes i the root of the tree:

$\mathbb{E}$	$\mathbb{T}$	$\mathbb{A}$	$\mathbb{G}$	$\mathbb{S}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$0 \xrightarrow{root} i$	$i \mapsto \{[\,]\}$			i

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 $\text{CHOOSE}(\tau, G)$ transition:

	$i \mapsto T$		$i \notin Dom(\mathbb{G})$	$\sigma \vert i$
	$i \mapsto \{\tau\}$	$i \mapsto \emptyset$	$i \mapsto G$	$\sigma \vert i$

This transition may only be applied if the specific term type $\tau \in T$ is apply-reachable from the newly selected lexical type τ_G. The CHOOSE operation is the only operation allowed when the active node does not have a graph constant yet; therefore, it always determines the lexical type of i *first*, before any outgoing edges are added.

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 APP_α edge to a node j that has no incoming edge, adding j to the stack:

$j \notin Dom(\mathbb{E})$	$i \mapsto \{\tau\}$	$i \mapsto A$	$i \mapsto G$	$\sigma \vert i$
$i \xrightarrow{\text{APP}_\alpha} j$	$j \mapsto \{req_\alpha(\tau_G)\}$	$i \mapsto A \cup \{\alpha\}$		$\sigma \vert i \vert j$

Here α must be a source in the apply set $\mathcal{A}(\tau_G, \tau)$ but not in $\mathbb{A}(i)$, i.e. be a source of G that still needs to be filled. Fixing the term type of j ensures the type restriction 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 MOD_β edge to a token j that has no incoming edge, also adding j to the stack:

$j \notin Dom(\mathbb{E})$	$i \mapsto \{\tau\}$	$i \mapsto A$	$i \mapsto G$	$\sigma \vert i$
$i \xrightarrow{\text{MOD}_\beta} j$	$j \mapsto T'$			$\sigma \vert i \vert j$

We require that T' is the set of all types $\tau' \in \Omega$ such that all sources in τ' (except β) including their requests are already present in τ_G, and $req_\beta(\tau') = [\,]$, reflecting constraints on the MOD operation in Groschwitz (2019).

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	INIT 3
3			$wants_3 \mapsto \emptyset$	$wants_3 \mapsto G_{\text{want}}$	3	CHOOSE $[]$, $\langle G_{\text{want}}, [s, o[s]]\rangle$
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 $[]$, $\langle G_{\text{writer}}, []\rangle$
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 $[s]$, $\langle G_{\text{sleep}}, [s]\rangle$
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 $[m]$, $\langle G_{\text{soundly}}, [m]\rangle$
11					[]	3 × POP

Figure 4: Derivation with LTF of the AM dependency tree in Fig. 2. 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.

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.

	$i \mapsto \{\tau\}$	$i \mapsto \mathcal{A}(\tau_G, \tau)$	$i \mapsto G$	$\sigma \vert i$
				σ

5.3 Lexical type first (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 left, 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 in §5.4 where we may not yet know the graph constant for i, we let $K_c(i) = \{\tau_{\mathbb{G}_c(i)}\}$ if $i \in Dom(\mathbb{G}_c)$ and $K_c(i) = \Omega$ otherwise. That is, if the graph constant $\mathbb{G}_c(i)$ is not yet defined, we assume we can choose it freely later. Then we can define

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

i.e. $O_c(i)$ is the minimal number of sources we need in addition to the ones already covered in $\mathbb{A}_c(i)$ in order to cover the apply set $\mathcal{A}(\lambda, \tau)$, assuming we choose the lexical type λ and term type τ optimally within the current constraints. 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 $PossL(\tau, A, n)$ 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, i.e. $|\mathcal{A}(\lambda, \tau) - A| \leq n$.

We prevent dead ends (see §5.5) by requiring that CHOOSE(τ, G) can only be applied to a configuration c if $\tau_G \in PossL(\tau, \emptyset, W_c - O_c)$. Then τ is apply-reachable from τ_G with at most $W_c - O_c$ APPLY transitions; this is exactly as many as we can spare. The MODIFY transition reduces the number of tokens that have no incoming edge without performing an APPLY transition, so we only allow it when we have tokens 'to spare', i.e. $W_c - O_c \geq 1$.

5.4 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.

APPLY and MODIFY. We modify the APPLY and MODIFY operations from §5.3 such that they no longer assign term types to children and do not push the child on the stack. This allows the transition system to add outgoing edges to the active node i without committing to types. The APPLY(α, j) transition becomes

$j \notin Dom(\mathbb{E})$	$i \mapsto T$	$i \mapsto A$		$\sigma \vert i$
$i \xrightarrow{\text{APP}_\alpha} j$		$i \mapsto A \cup \{\alpha\}$		$\sigma \vert i$

Because we do not yet know the types for i and thus neither the apply set $\mathcal{A}(\lambda, \tau)$, we cannot directly check that this APPLY transition will not lead to a dead end. Instead, we check if there are *possible* types τ and λ with α in their apply set, by requiring that $\bigcup_{\tau \in T} PossL(\tau, A \cup \{\alpha\}, W_c - 1)$ is

non-empty (it is $W_c - 1$ to account for the edge we are about to add). We also keep the restriction that $\alpha \notin A$, to avoid duplicate APP_α edges.

The $\text{MODIFY}(\beta, j)$ transition becomes

$j \notin Dom(\mathbb{E})$				$\sigma \vert i$
$i \xrightarrow{\text{MOD}_\beta} j$				$\sigma \vert i$

Again, 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.

$i \xrightarrow{\text{APP}_{\alpha_k}} j_k$ $i \xrightarrow{\text{MOD}_{\beta_k}} l_k$	$i \mapsto T$	$i \mapsto A$		$\sigma \vert i$
	$i \mapsto \{\tau\}$, $j_k \mapsto T_k$, $l_k \mapsto T'_k$	$j_k \mapsto \emptyset$, $l_k \mapsto \emptyset$	$i \mapsto G$	$\sigma \vert l_1 \vert \ldots \vert l_r$ $\vert j_1 \vert \ldots \vert j_s$

$\text{FINISH}(G)$ is allowed if $\mathcal{A}(\tau_G, \tau) = A$ for some $\tau \in T$, and fixes this τ as the term type. In addition, FINISH pushes the child nodes j_k of all $s \geq 0$ outgoing APP edges onto the stack and fixes their term types as $T_k = \{req_{\alpha_k}(\tau_G)\}$ (like in APPLY of §5.2). Similarly, FINISH also pushes the child nodes l_k of all $r \geq 0$ outgoing MOD edges onto the stack and computes their term type sets T'_k as in the MODIFY rule of §5.2. We push the children in the reverse order of when they were created, so that they are popped off the stack in the order the edges were drawn.

Finally, since CHOOSE no longer exists, we must set $\mathbb{A}(i) = \emptyset$ during INIT. An example run is shown in Appendix F.

5.5 Correctness

We state the main correctness results here; proofs are in Appendix G. We assume for all types $\lambda \in \Omega$ and all sources $\alpha \in S$, that the type $req_\alpha(\lambda)$ is also in Ω, and that for every source β with $\text{MOD}_\beta \in L$, the type $[\beta]$ is in Ω. This allows us to select lexical types that do not require unexpected APP children.

Theorem 5.1 (Soundness). Every goal configuration derived by LTF or LTL corresponds to a well-typed AM dependency tree.

Theorem 5.2 (Completeness). For every well-typed AM dependency tree t, there are sequences of LTF and LTL transitions that build t.

Theorem 5.3 (No dead ends). Every configuration derived by LTF or LTL can be completed to a goal configuration.

5.6 Neural model

We train a neural model to predict LTF and LTL transitions, by extending Ma et al.'s stack-pointer model with means to predict graph constants and term types. We phrase AM dependency parsing as finding the most likely sequence $d^{(1)}, \ldots, d^{(N)}$ of LTF or LTL transitions given an input sentence $\mathbf{x}$, factorized as follows:

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

We encode the sentence with a multi-layer BiLSTM based on embeddings for word, POS tag, lemma, named entity tag and character CNN, yielding a sequence of hidden states $\mathbf{s}_1, \ldots \mathbf{s}_n$. The decoder LSTM is initialized with the last hidden state of the encoder and is updated as follows:

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

where tos denotes the node on top of the stack, p the parent of tos and c refers to the most recently generated child of tos. Let further $a_i^{(t)} \propto \exp \text{Biaffine}(\mathbf{h}^{(t)}, \mathbf{s}_i)$ be an attention score and let $\mathbf{s}'$ be a second BiLSTM encoding trained to predict graphs and term types.

When in the start configuration, the probability of INIT selecting node i as the root is $P(\text{INIT } i | \mathbf{h}^{(t)}) = a_i^{(t)}$; otherwise it is zero.

In LTF, if after $d^{(1)}, \ldots, d^{(t-1)}$ the CHOOSE transition is allowed (and thus required), we have the transition probabilities

$$P_\theta(\text{CHOOSE } (\tau, G) | \mathbf{h}^{(t)}) = P_\theta(\tau | \mathbf{h}^{(t)}) \cdot P_\theta(G | \mathbf{h}^{(t)})$$

where we score the graph constant G and term type τ with softmax functions

$$P_\theta(G | \mathbf{h}^{(t)}) = \text{softmax}(\text{MLP}^G([\mathbf{h}^{(t)}, \mathbf{s}'_{tos}]))_G$$
$$P_\theta(\tau | \mathbf{h}^{(t)}) = \text{softmax}(\text{MLP}^{tt}([\mathbf{h}^{(t)}, \mathbf{s}'_{tos}]))_\tau.$$

In this situation, the probabilities of all other transitions are 0.

In contrast, if in LTF the CHOOSE transition is not allowed, we can draw an edge or POP. We score the target j of the outgoing edge with the attention score $a_j^{(t)}$ and model the probability for POP with an artificial word at position 0 (using an attention score $a_0^{(t)}$). In other words, we have

$$P_\theta(\ell, j | \mathbf{h}^{(t)}) = a_j^{(t)} \cdot P_\theta(\ell | \mathbf{h}^{(t)}, tos \to j)$$
$$P_\theta(\text{POP} | \mathbf{h}^{(t)}) = a_0^{(t)}$$

where we score the edge label ℓ with a softmax:

$$P_\theta(\ell|\mathbf{h}^{(t)}, tos \to j) = \text{softmax}(\text{MLP}^{lbl}([\mathbf{h}^{(t)}, \mathbf{s}_j]))_\ell.$$

In this situation, CHOOSE has probability 0.

In LTL, we must decide between drawing an edge and FINISH; we score edges as in LTF and replace the probabilities for CHOOSE and POP with

$$P_\theta(\text{FINISH}(G)|\mathbf{h}^{(t)}) = a_0^{(t)} P_\theta(G|\mathbf{h}^{(t)})$$

where $P_\theta(G|\mathbf{h}^{(t)})$ is as above.

Training. The training objective is MLE of θ on a corpus of AM dependency trees. There are usually multiple transition sequences that lead to the same AM dependency tree, so we follow Ma et al. and determine a canonical sequence by visiting the children in an inside-out manner.

Inference. During inference, we first decide whether we have to CHOOSE. If not, we divide each transition into two greedy decisions: we first decide, based on $a_i^{(t)}$, whether to FINISH/POP or whether to add an edge (and where); second we find the graph constant (in case of FINISH) or the edge label. To ensure well-typedness, we set the probability of forbidden transitions to 0.

Run-time complexity. The run-time complexity of the parser is $O(n^2)$: $O(n)$ transitions, each of which requires evaluating attention over n tokens.

The code is available at `https://github.com/coli-saar/am-transition-parser`.

6 Evaluation

Data. We evaluate on the DM, PAS, and PSD graphbanks from the SemEval 2015 shared task on Semantic Dependency Parsing (SDP, Oepen et al. (2015)), the EDS corpus (Flickinger et al., 2017) and the AMRBank releases LDC2015E86, LDC2017T10 and LDC2020T02 (Banarescu et al., 2013). We use the AM dependency tree decompositions of these corpora from Lindemann et al. (2019) (L'19 for short) as training data, as well as their pre- and post-processing pipeline (including the AMR post-processing bugfix published after submission). We use the same hyperparameters and hardware for all experiments (see Appendices B and C).

Baselines. We compare against the fixed tree and projective decoders of Groschwitz et al. (2018), using costs computed by the model of L'19. For the projective decoder we train with the edge existence loss recommended by Groschwitz et al. (2018). The models use pretrained BERT embeddings (Devlin et al., 2019) without finetuning.

6.1 A* parsing

Table 1 compares the parsing accuracy of the A* parser (with the cost model of the projective parser) across the six graphbanks (averaged over 4 training runs of the model), with the Init rule restricted to the six lowest-cost graph constants per token. We only report one accuracy for A* because A* search is optimal, and thus the accuracies with different admissible heuristics are the same. As expected, the accuracy is on par with L'19's parser; it is slightly degraded on DM, EDS and AMR, perhaps because these graphbanks require non-projective AM dependency trees for accurate parsing.

Parsing times are shown in Table 2 as tokens per second. We limit the number of items that can be dequeued from the agenda to one million per sentence. This makes two sentences per AMR test set unparseable; they are given dummy single-node graphs for the accuracy evaluation. The A* parser is significantly faster than L'19's fixed-tree decoder; even more so than the projective decoder on which it is based, with a 10x to 1000x speedup. Each SDP test set is parsed in under a minute.

The speed of the A* parser is very sensitive to the accuracy of the suppertagging model: if the parser takes many supertags for a token off the agenda before it finds the goal item for a well-typed tree, it will typically deqeueue many items altogether. On the SDP corpora, the supertagging accuracy on the dev set is above 90%; here even the trivial heuristic is fast because it simply dequeues the best supertag for most tokens. On AMR, the supertagging accuracy drops to 78%; as a consequence, the A* parser is slower overall, and the more informed heuristics yield a higher speedup. EDS is an outlier, in that the supertagging accuracy is 94%, but the parser still dequeues almost three supertags per token on average. Why this is the case bears further study.

6.2 Transition-based parsing

To evaluate the transition-based parser, we extract the graph lexicon and the type set Ω from the training and development sets such that Ω includes all lexical types and term types used. We establish the assumptions of §5.5 by automatically adding up to 14 graph constants per graphbank, increasing the graph lexicon by less than 1%.

The LTL parser is accurate with greedy search and parses each test set in under a minute on the CPU and within 20 seconds on the GPU[3]. Since

[3] See Lindemann (2020) for the GPU implementation.

	DM		PAS		PSD		EDS		AMR 15	AMR 17	AMR 20
	id F	ood F	id F	ood F	id F	ood F	Smatch F	EDM	Smatch F	Smatch F	Smatch F
He and Choi (2020)♠	**94.6**	90.8	**96.1**	**94.4**	**86.8**	79.5	-	-	-	-	-
Chen et al. (2018)	-	-	-	-	-	-	**90.9**	**90.4**	-	-	-
Cai and Lam (2020)♠	-	-	-	-	-	-			-	**80.2**	-
Zhang et al. (2019)♠	92.2	87.1	-	-	-	-	-	-	-	77.0±0.1	-
FG'20♠	94.4	**91.0**	95.1	93.4	82.6	**82.0**	-	-	-	-	-
L'19♠, w/o MTL	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	75.4±0.1	76.3±0.2	75.2±0.1
A* parser♠	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	74.5±0.1	75.3±0.1	74.5±0.1
LTL♠, no 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	45.7±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	74.9±0.3	76.5±0.1	76.0±0.1
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	**75.7**±0.3	**77.1**±0.1	**76.8**±0.1
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	71.4±0.2	73.2±0.4	72.6±0.2
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	74.8±0.2	76.1±0.2	75.3±0.4

Table 1: Semantic parsing accuracies (id = in domain test set; ood = out of domain test set). ♠ marks models using BERT. L'19 are results of Lindemann et al. (2019) with fixed tree decoder (incl. post-processing bugfix). FG'20 is Fernández-González and Gómez-Rodríguez (2020).

	DM	PAS	PSD	EDS	A15	A17	A20
projective, L'19♠costs	3	2	4	4	<2	<2	<2
L'19♠fixed tree	710	97	265	542	<4	<3	<3
A*♠, trivial	706	2096	1235	105	<9	<10	<6
A*♠, edge-based	725	2105	1421	129	<20	<26	<20
A*♠, ignore-aware	712	2167	1318	136	<22	<30	<26
LTL♠, GPU, greedy	**4,750**	**4,570**	**2,742**	**4,443**	**1,977**	**2,116**	**1,946**
LTL♠, CPU, greedy	1,094	913	1,126	968	879	962	865
beam=3	241	203	231	217	217	205	198
LTF♠, CPU, greedy	852	791	688	673	563	424	514
beam=3	145	123	96	108	100	76	78

Table 2: Avg. parsing speed in tokens/s on test sets. < indicates where parsing was interrupted due to timeout.

the BERT embeddings take considerable time to compute, parsing without BERT leads to a parsing speed of up to 10,000 tokens per second (see Appendix A). With beam search, LTL considerably outperforms L'19 on AMR, matching the accuracy of the fast parser of Zhang et al. (2019) on AMR 17 while outperforming it by up to 3.3 points F-score on DM. On the other graphbanks, LTL is on par with L'19. When evaluated without BERT, LTL outperforms L'19 by more than 1 point F-score on most graphbanks (see Appendix A).

The LTF parser is less accurate than LTL. Beam search reduces or even closes the gap, perhaps because it can select a better graph constant from the beam after selecting edges.

Note that accuracy drops drastically for a variant of LTL which does not enforce type constraints ("LTL, no types") because up to 50% of the predicted AM dependency trees are not well-typed and cannot be evaluated to a graph. The neural model does not learn to reliably construct well-typed trees by itself; the type constraints are crucial.

Overall, the accuracy of LTL is very similar to L'19, except for AMR where LTL is better. We investigated this difference in performance on AMR 17 and found that LTL achieves higher precision but its recall is worse for longer sentences[4]. We suspect this is because LTL is not explicitly penalized for leaving words out of the dependency tree and thus favors shorter transition sequences.

7 Conclusion

We have presented two fast and accurate algorithms for AM dependency parsing: an A* parser which optimizes Groschwitz et al.'s projective parser, and a novel transition-based parser which builds an AM dependency top-down while avoiding dead ends.

The parsing speed of the A* parser differs dramatically for the different graphbanks. In contrast, the parsing speed with the transition systems is less sensitive to the graphbank and faster overall. The transition systems also achieve higher accuracy.

In future work, one could make the A* parser more accurate by extending it to non-projective dependency trees, especially on DM, EDS and AMR. The transition-based parser could be made more accurate by making bottom-up information available to its top-down choices, e.g. with Cai and Lam's (2020) "iterative inference" method. It would also be interesting to see if our method for avoiding dead ends can be applied to other formalisms with complex symbolic restrictions.

Acknowledgments. We thank the anonymous reviewers and the participants of the DELPH-IN Summit 2020 for their helpful feedback and comments. We thank Rezka Leonandya for his work on an earlier version of the A* parser. This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), project KO 2916/2-2.

[4]For both parsers we model the dependence of recall on sentence length with linear regression; the slopes of the two models are significantly different, $p < 0.05$.

References

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*.

Tatiana Bladier, Jakub Waszczuk, Laura Kallmeyer, and Jörg Hendrik Janke. 2019. From partial neural graph-based LTAG parsing towards full parsing. *Computational Linguistics in the Netherlands Journal*, 9:3–26.

Jan Buys and Phil Blunsom. 2017. Robust incremental neural semantic graph parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*.

Deng Cai and Wai Lam. 2020. AMR parsing via graph-sequence iterative inference. In *Proceedings of the ACL*.

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 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 408–418, Melbourne, Australia. Association for Computational Linguistics.

Marco Damonte, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for Abstract Meaning Representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1*.

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 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

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*.

Rebecca Dridan and Stephan Oepen. 2011. Parser evaluation using elementary dependency matching. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 225–230.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent Neural Network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Federico Fancellu, Sorcha Gilroy, Adam Lopez, and Mirella Lapata. 2019. Semantic graph parsing with recurrent neural network DAG grammars. In *Proceedings of the EMNLP-IJCNLP*.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2020. Transition-based semantic dependency parsing with pointer networks. In *Proceedings of the ACL*.

Dan Flickinger, Jan Hajič, Angelina Ivanova, Marco Kuhlmann, Yusuke Miyao, Stephan Oepen, and Daniel Zeman. 2017. Open SDP 1.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Jonas Groschwitz. 2019. *Methods for taking semantic graphs apart and putting them back together again*. Ph.D. thesis, Macquarie University and Saarland University.

Jonas Groschwitz, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2017. A constrained graph algebra for semantic parsing with AMRs. In *Proceedings of the 12th International Conference on Computational Semantics (IWCS)*.

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*.

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.

Dan Klein and Christopher D. Manning. 2003. A* parsing: fast exact viterbi parse selection. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*.

Alexander Koller, Stephan Oepen, and Weiwei Sun. 2019. Graph-based meaning representations: Design and processing. Tutorial at ACL 2019.

Mike Lewis and Mark Steedman. 2014. A* CCG parsing with a supertag-factored model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 990–1000, Doha, Qatar. Association for Computational Linguistics.

Matthias Lindemann. 2020. Fast transition-based AM dependency parsing with well-typedness guarantees. MSc thesis, Saarland University.

Matthias Lindemann, Jonas Groschwitz, and Alexander Koller. 2019. Compositional semantic parsing across graphbanks. 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*.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

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)*.

Tianze Shi and Lillian Lee. 2018. Valency-augmented dependency parsing. In *Proceedings of the EMNLP*.

Stuart Shieber, Yves Schabes, and Fernando Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

Gisle Ytrestøl. 2011. Optimistic backtracking: a backtracking overlay for deterministic incremental parsing. In *Proceedings of the ACL 2011 Student Session*. Association for Computational Linguistics.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019. Broad-coverage semantic parsing as transduction. In *Proceedings of the EMNLP-IJCNLP*.

Yue Zhang and Stephen Clark. 2011. Shift-reduce ccg parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, USA.

A Additional experiments and dev accuracies

Table 3 shows the results of further experiments (means and standard deviations over 4 runs). Models that do not use BERT, use GloVe embeddings of size 200. Note that we use the pre- and post-processing of Lindemann et al. (2019) in the most recent version, which fixes a bug in AMR post-processing[5].

For each model trained, Table 4 shows the performance of one run on the development set.

Table 6 shows F-scores of different versions of Smatch on the AMR tests. See also Appendix E.

B Hardware and parsing experiments

All parsing experiments were performed on Nvidia Tesla V100 graphics cards and Intel Xeon Gold 6128 CPUs running at 3.40 GHz.

We measure run-time as the sum of the GPU time and the CPU time on a single core for all approaches. When computing scores for A*, we use a batch size of 512 for all graphbanks but AMR, where we use a batch size of 128. We use a batch size of 64 for LTL and LTF for parsing on the CPU. The transition probabilites are computed on the GPU and then transferred to the main memory. In the parsing experiments with LTL where the transition system is run on the GPU as well, we use a batch size of 512, except for AMR, for which we use a batch size of 256.

The A* algorithm is implemented in Java and was run on the GraalVM 20 implementation of the JVM.

We run the projective parser and the fixed tree parser of Groschwitz et al. (2018) with the 6 best 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 use a dummy graph with a single node.

LTL and LTF are implemented in python and run on CPython version 3.8.

C Hyperparameters and training details

C.1 Scores for A*

We obtain the scores by training the parser of Lindemann et al. (2019). Since Groschwitz et al. (2018) argue that a hinge loss such as the one that L'19 use might not be well-suited for the projective parser, we replaced it by the log-likelihood loss of

[5] see https://github.com/coli-saar/am-parser

	DM		PAS		PSD		EDS		AMR 15	AMR 17	AMR 20
	id F	ood F	id F	ood F	id F	ood F	Smatch F	EDM	Smatch F	Smatch F	Smatch F
L'19 + CharCNN	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	70.5±0.4	71.7±0.2	70.4±0.5
L'19♠+ CharCNN	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	75.4±0.2	76.4±0.1	74.8±0.2
A* parser♠	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	74.5±0.1	75.3±0.1	74.5±0.1
LTL♠, no 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	45.7±1.0
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	71.1±0.2	73.1±0.2	72.5±0.2
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	71.8±0.3	73.7±0.3	73.3±0.3
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	74.9±0.3	76.5±0.1	76.0±0.1
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	**75.7**±0.3	**77.1**±0.1	**76.8**±0.1
LTF♠, no 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	37.6±0.7
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	67.7±0.2	69.2±0.3	69.0±0.2
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	71.0±0.2	72.2±0.2	72.2±0.2
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	71.4±0.2	73.2±0.4	72.6±0.2
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	74.8±0.2	76.1±0.2	75.3±0.4

Table 3: Fulls details of accuracies of parsers we have trained (id = in domain test set; ood = out of domain test set). ♠ marks models using BERT. L'19 is Lindemann et al. (2019) with fixed tree decoder.

	DM	PAS	PSD	EDS		AMR 15	AMR 17	AMR 20
	F	F	F	Smatch	EDM	Smatch	Smatch	Smatch
L'19 + charCNN	91.2	91.7	80.6	88.6	84.1	71.6	72.9	73.0
L'19♠+ charCNN	94.2	95.0	84.3	90.6	86.0	75.9	77.3	77.5
A*♠	92.1	94.6	84.0	88.0	83.8	75.1	76.4	76.9
LTL, greedy	92.1	92.9	80.8	89.1	84.6	72.7	74.7	75.6
LTL♠, greedy	94.1	95.1	83.4	90.6	85.9	76.1	78.0	78.4
LTF, greedy	91.0	92.4	75.7	87.1	82.8	69.2	70.7	72.1
LTF♠, greedy	92.9	94.59	79.7	88.7	84.2	72.8	74.8	75.4

Table 4: Results on development sets.♠ marks models using BERT. L'19 is Lindemann et al. (2019) with fixed tree decoder.

	DM	PAS	PSD	EDS	AMR 15	AMR 17	AMR 20
LTL, greedy	1,180	1,128	1,288	1,154	1,121	1,162	1,148
LTL, beam=3	257	229	243	224	234	222	210
LTL, GPU, greedy	**10,266**	**10,271**	**4,201**	**9,188**	**3,647**	**3,413**	**2,912**
LTF, greedy	957	908	755	752	672	578	572
LTF, beam=3	153	126	97	113	104	91	81

Table 5: Avg. parsing speed of transition systems in tokens/s on (in-domain) test sets without BERT. For result with BERT, see main paper.

Groschwitz et al. (2018). The development metric based on which the model is chosen is the arithmetic mean between supertagging accuracy and labeled attachment score.

We follow Lindemann et al. (2019) in the hyperparamters, with two exceptions: we use batch size of 32 instead of 64 because of memory constraints and we add a character CNN to the model to make it more comparable with the model of the transition systems; see below for its hyperparameters. In order to tease apart the impact of the character CNN, we include the performance of a L'19 model with the character CNN in Table 3. Differences are within one standard deviation of the results obtained with the original architecture used in L'19.

	AMR 2015		AMR 2017	
	new F	L'19 F	new F	L'19 F
L'19 + CharCNN	70.5±0.4	70.2±0.4	71.7±0.2	71.4±0.2
L'19♠+ CharCNN	75.4±0.2	75.1±0.2	76.4±0.1	76.1±0.1
L'19♠, w/o MTL	75.4±0.1	75.1±0.2	76.3±0.2	76.0±0.2
A* parser♠	74.5±0.1	74.2±0.1	75.3±0.1	75.1±0.1
LTL, greedy	71.1±0.2	70.7±0.2	73.1±0.2	72.8±0.3
beam = 3	71.8±0.3	71.4±0.3	73.7±0.3	73.4±0.3
LTL♠, greedy	74.9±0.3	74.5±0.3	76.5±0.1	76.3±0.1
beam=3	75.7±0.3	75.3±0.3	77.1±0.1	76.8±0.1
LTF, greedy	67.7±0.2	67.3±0.2	69.2±0.3	68.9±0.2
beam = 3	71.0±0.2	70.6±0.2	72.2±0.2	71.9±0.2
LTF♠, greedy	71.4±0.2	71.1±0.1	73.2±0.4	72.9±0.4
beam=3	74.8±0.2	74.5±0.2	76.1±0.2	75.8±0.2

Table 6: Results on AMR test sets with different versions of Smatch. L'19 F is the version that was used by Lindemann et al. (2019) and *new F* is version 1.0.4.

C.2 LTL and LTF

We set the hyperparameters manually without extensive hyperparameter search, mostly following Ma et al. (2018). We followed Lindemann et al. (2019) for number of hidden units and dropout in the MLPs for predicting graph constants and for the size of embeddings.

We follow Lindemann et al. (2019) in splitting 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 Adam 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 don't perform weight decay or gradient clipping. In experiments with GloVe, we use the vectors of dimensionality 200 (6B.200d) and fine-tune them. Following Ma et al. (2018), we employ a character CNN with 50 filters of window size 3.

We use the `BERT large` version of BERT and

POS	32
Characters	100
NE embedding	16

Table 7: Dimensionality of embeddings used in all experiments.

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

Table 8: Hyperparameters of LTL and LTF

average the layers. The weights for the average are learned but we do not fine-tune BERT itself.

For the second encoding of the input sentence, $\mathbf{s}'$, we use a single-layer bidirectional LSTM when using BERT and a two-layer bidirectional LSTM when using GloVe. On top of $\mathbf{x}'$ we perform variational dropout with $p = 0.33$, as well as on top of $\mathbf{s}$ and $\mathbf{s}'$. The other hyperparameters are listed in Tables 7, 8 and 9. The number of parameters of the LTL and LTF models are in table 10.

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

D Data

We use the AM dependency trees of Lindemann et al. (2019) as training data, along with their preprocessing. See their supplementary materials for more details. For completeness, Table 11 shows the number of AM dependency trees in the training sets as well as the number of sentences and tokens in the test sets. Note that the heuristic approach cannot obtain AM dependency trees for all graphs in the training data but nothing is left out of the test data.

Layers	1
Hidden units	1024
Activation	tanh
Dropout	0.4

Table 9: Hyperparameters used in MLPs for predicting delexicalized constants, term types and lexical labels.

	LTL		LTF		L'19	
	GloVe	BERT	GloVe	BERT	Glove	BERT
DM	67.39	61.77	69.59	63.97	19.19	8.76
PAS	66.71	61.05	68.90	63.24	18.54	8.05
PSD	73.95	68.15	76.40	70.60	25.84	15.15
EDS	70.35	65.98	72.59	68.23	21.52	12.97
AMR 15	71.49	68.34	73.88	70.73	22.07	15.34
AMR 17	76.42	71.60	78.86	74.04	27.84	18.61
AMR 20	82.63	75.56	85.13	78.06	35.16	22.33

Table 10: Number of trainable parameters (including GloVe embeddings) in millions.

We use the standard splits on all data sets into training/dev/test, again following Lindemann et al. (2019).

PAS, PSD and AMR are licensed by LDC but the DM and EDS data can be downloaded from http://hdl.handle.net/11234/1-1956.

E Evaluation metrics

DM, PAS and PSD We compute labeled F-score with the evaluation toolkit that was developed for the shared task: https://github.com/semantic-dependency-parsing/toolkit.

EDS We evaluate with Smatch Cai and Knight (2013), in this implementation due to its high speed: github.com/Oneplus/tamr/tree/master/amr_aligner/smatch and EDM (Dridan and Oepen, 2011) in the implementation of Buys and Blunsom (2017): https://github.com/janmbuys/DeepDeepParser. We follow Lindemann et al. (2019) in using Smatch as development metric.

	Training		Test	
	Sentences	AM dep. trees	Sentences	Tokens
DM	35,657	31,349	1,410	33,358
PAS	35,657	31,796	1,410	33,358
PSD	35,657	32,807	1,410	33,358
EDS	33,964	25,680	1,410	32,306
AMR 15	16,833	15,472	1,371	28,458
AMR 17	36,521	33,406	1,371	28,458
AMR 20	55,635	51.515	1,898	36,928

Table 11: Data statics after preprocessing.Test set is in-domain for SDP.

AMR We evaluate with Smatch in the original implementation https://github.com/snowblink14/smatch. In the main paper, we report results with Smatch 1.0.4, which are somewhat better than with earlier versions. This also applies to the results of Lindemann et al. (2019). Table 6 shows results with the Smatch version that were originally used in Lindemann et al. (2019) (Commit `ad7e65` from August 2018).

F Example for LTL

Fig. 5 shows an example of a derivation with LTL, analogous to the one in Fig. 4.

G Proofs

The proofs given here follow exactly Lindemann (2020).

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. It will turn out that significant assumptions are only needed to guarantee that there are no dead ends.

Throughout this section we assume the type system of (Groschwitz, 2019), where types are formally defined as DAGs with sources as nodes, and requests being defined via the edges.

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

Lemma G.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 G.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 G.1. □

G.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 G.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 G.4. In every configuration c derived by LTF or LTL, token i has an APP_α child if and only if $\alpha \in \mathbb{A}_c(i)$.

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 5: Derivation with LTL of the AM dependency tree in Fig. 2. 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.

Proof. The APPLY(α, j) transitions in LTF and LTL always add an α-source to $\mathbb{A}_c(i)$ and simultaneously add an APP$_\alpha$ edge. There are no other ways to add a source to $\mathbb{A}_c(i)$ or to create an APP$_\alpha$ 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 G.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 G.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 APP$_{\alpha_1}, \ldots,$ APP$_{\alpha_n}$, respectively. Let i also have MOD children $m_1, \ldots, m_k$, attached with the edges MOD$_{\beta_1}, \ldots,$ 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$_\alpha$ child if and only if $\alpha \in \mathcal{A}(\lambda, \tau)$

(ii) if a is an APP$_\alpha$ child of i, then it has the term type $\mathit{req}_\alpha(\lambda)$.

(i) follows from the goal condition $\mathbb{A}_c(i) = \mathcal{A}(\lambda, \tau)$ and Lemma G.4.

(ii) the only way the edge $i \xrightarrow{\text{APP}_\alpha} a$ can be created is by the APPLY(α, a) transitions with i on top of the stack. Both transition systems enforce $\mathbb{T}_c(a) = \{\mathit{req}_\alpha(\lambda)\}$. Using the inductive hypothesis on a, it follows that a evaluates to a graph of type $\mathit{req}_\alpha(\lambda)$.

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$_\beta$ child m that evaluates to a graph of type τ' by the inductive hypothesis, we have to show that $\tau' - \beta \subseteq \lambda \wedge \mathit{req}_\beta(\tau') = []$. The MOD$_\beta$ edge was created by a MODIFY(β, m) transition. The MODIFY(β, 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: $\mathbb{T}_{c'}(m) = \{\tau \in \Omega | \tau - \beta \subseteq \lambda \wedge \mathit{req}_\alpha(\tau) = []\}$. 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$_\beta$ edge was created and c is a goal configuration), which resulted in $\mathbb{T}_c(m) = \{\tau'\}$, where $\tau' \in \mathbb{T}_{c'}(m) = \{\tau \in \Omega | \tau - \beta \subseteq \lambda \wedge \mathit{req}_\alpha(\tau) = []\}$. This means that the well-typedness condition indeed also holds for τ'. □

G.2 Completeness

Theorem G.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 1 and 2 give transition sequences that, when prefixed with an appropriate INIT operation, generate t. We show this by showing the following lemma

(for LTF):

Lemma G.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 1) 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 G.6 for LTF then follows from applying the lemma to the whole tree t and the configuration obtained after $\textsc{Init}(t)$. This yields a configuration with empty stack, which is a goal configuration (see Lemma G.1).

Before we approach the proof of Lemma G.7, we need to show the following:

Lemma G.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 G.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 $\tau_G \in \mathit{PossL}(\tau_G, \emptyset, W_c - O_c)$, 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 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 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 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 in Fig. 6, 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

$$c \xrightarrow{\text{CHOOSE}(\tau, G)} c'_0 \xrightarrow{\text{APPLY}(\alpha_1, a_1)} c_1 \xrightarrow{H_{LTF}(a_1, c_1)} c'_1 \dots c'_{n-1} \xrightarrow{\text{APPLY}(\alpha_n, a_n)} c_n \xrightarrow{H_{LTF}(a_n, c_n)} c_{n'}$$
$$c'_n \xrightarrow{\text{MODIFY}(\beta_1, m_1)} c_{n+1} \xrightarrow{H_{LTF}(m_1, c_{n+1})} c'_{n+1} \dots \xrightarrow{H_{LTF}(m_k, c_{n+k})} c'_{n+k} \xrightarrow{\text{POP}} c' \quad (1)$$

Figure 6: Transition sequence returned by $H_{LTF}(t, c)$ in the induction step.

edges from i to $a_1, \dots, a_n, m_1, \dots, m_k$ with the correct labels (consequence (a)) and also makes the assignment $\mathbb{G}_{c'}(i) = G_i$ using $\text{CHOOSE}(\tau, G)$ (consequence (b)).

Now we go over the transition sequence in Eq. 6 and check that the transitions can be applied, the conditions (i)-(v) hold and what happens to the stack.

First, in order for $\text{CHOOSE}(\tau_G, \tau)$ 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 $\tau_G \in \mathit{PossL}(\tau, \emptyset, W_c - O_c)$, which is equivalent to

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

Since $\mathcal{A}(\tau_G, \tau) = \{\alpha_1, \dots, \alpha_n\}$ and $W_c - O_c \geq |t| - 1 \geq |\{\alpha_1, \dots, \alpha_n\}| = n$, 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 $\text{APPLY}(\alpha_1, a_1)$. 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 $\mathbb{A}_{c_1}(i) = \{\alpha_1\}$, $\mathbb{T}_{c_1}(a_1) = \{\mathit{req}_{\alpha_1}(\tau_G)\}$ 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 $\mathbb{T}_{c_1}(a_1) = \{\mathit{req}_{\alpha_1}(\tau_G)\} = \{\tau_{a_1}\}$ 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 $j \notin \mathit{Dom}(\mathbb{G}_c)$ for all nodes j of t and $j \notin \mathit{Dom}(\mathbb{E}_c)$ 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 G.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

$$|t| - 1 \geq n + \sum_{j=1}^{n} (|a_j| - 1) \quad (2)$$

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. 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

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

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 2.

Algorithm 1 Generate LTF transitions for AM dependency tree

function $H_{LTF}(c, t)$
  Let t have graph constant G
  and term type τ
  $c \leftarrow \text{CHOOSE}(\tau, G)(c)$
  for APP_α child a of t **do**
    $c \leftarrow \text{APPLY}(\alpha, a)(c)$
    $c \leftarrow H_{LTF}(c, a)$
  end for
  for MOD_β child m of t **do**
    $c \leftarrow \text{MODIFY}(\beta, m)(c)$
    $c \leftarrow H_{LTF}(c, m)$
  end for
  $c \leftarrow \text{POP}(c)$
  return c
end function

Algorithm 2 Generate LTL transitions for AM dependency tree

function $H_{LTL}(c, t)$
  Let t have graph constant G
  for APP_α child a of t **do**
    $c \leftarrow \text{APPLY}(\alpha, a)(c)$
  end for
  for MOD_β child m of t **do**
    $c \leftarrow \text{MODIFY}(\beta, m)(c)$
  end for
  $c \leftarrow \text{FINISH}(G)(c)$
  Let $t_1, \ldots t_n$ be the children of t
  on the stack in c
  for $i \in 1, \ldots, n$ **do**
    $c \leftarrow H_{LTL}(c, t_i)$
  end for
  return c
end function

G.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 G.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 G.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 how the transitions are scored. The

proof of Theorem G.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 G.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 S, a set of types Ω and a set of edge labels L. We now make very explicit 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 $\alpha \in S$, if $\mathit{req}_\alpha(\lambda)$ is defined then $\mathit{req}_\alpha(\lambda) \in \Omega$.

Assumption 3. If $\text{MOD}_\alpha \in \mathit{Lab}$ then $[\alpha] \in \Omega$.

Assumption 4. For all source names $\alpha \in S$, $\text{APP}_\alpha \in \mathit{Lab}$.

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

The assumptions made are almost perfectly met in practice, see the main paper.

In the proof of Theorem G.9 we want to use the fact $[\,] \in \Omega$; this follows from the assumptions above:

Lemma G.10. The empty type $[\,] \in \Omega$.

Proof. Assumption 2 says that for all types $\lambda \in \Omega$ and all sources $\alpha \in S$, the type $\mathit{req}_\alpha(\lambda)$ (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, $\mathit{req}_n(\tau) = [\,]$. □

G.3.1 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 G.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 G.8 and by definition of *PossL*, we have that $O_c(i) = 0$, so

$$O_{c'} = O_c + O_{c'}(i) \tag{3}$$

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 $\tau_G \in \mathit{PossL}(\tau, \emptyset, W_c - O_c)$, 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, 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 5.3, we made the restriction that MODIFY is only applicable if

$$W_c - O_c \geq 1 \tag{4}$$

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. 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 G.1 this means that c is a goal configuration. We empty the stack by repeatedly applying Algorithm 3.

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 G.12. For any configuration c, $C_{LTF}(c)$ (Algorithm 3) 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 3 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 G.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 G.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 $\text{APP}_{a_i} \in L$ 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 G.1, this is a goal configuration.

G.3.2 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 4) that produces a valid sequence of transitions that we repeatedly apply to reach a goal configuration.

Lemma G.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 3 Complete LTF sequence

function $C_{LTF}(c)$
  if $c = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$ **then**
    Let $G \in C$ with $\tau_G = [\,]$.
    return $\textsc{Init}(1), \textsc{Choose}([\,], G), \textsc{Pop}$
  end if
  if $\mathbb{S}_c = []$ **then return** []
  end if
  Let i be top of $\mathbb{S}_c$.
  if $O_c(i) = 0$ **then**
    if $i \in Dom(\mathbb{G}_c)$ **then**
      return $\textsc{Pop}$
    else
      Let $G \in C, \tau_G \in \mathbb{T}_c(i)$.
      return $\textsc{Choose}(\tau_G, G), \textsc{Pop}$
    end if
  end if
  Let $o = \mathcal{A}(\mathbb{G}_c(i), \tau) - \mathbb{A}_c(i)$
  where $\mathbb{T}_c(i) = \{\tau\}$
  Let $\rho_j = req_{o_j}(\tau_{\mathbb{G}_c(i)})$
  Let $a_1, \ldots, a_{|o|}$ be tokens without heads
  $s = []$
  for $a_j \in a_1, \ldots, a_{|o|}$ **do**
    Let G be a constant of type ρ_j
    $s = s \oplus \textsc{Apply}(o_j, a_j), \textsc{Choose}(\rho_j, G), \textsc{Pop}$
  end for
  return $s \oplus \textsc{Pop}$
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 G.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 preconditions must be fulfilled, i.e.

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

Expanding the definition of *PossL* we get:

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

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 $O_{c'}(i) \leq W_{c'}$.

$\textsc{Modify}(\beta, j)$ 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 $\textsc{Modify}(\beta, j)$ transition.

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

- i has an incoming $\textsc{App}_\alpha$ 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 $\textsc{Mod}_\beta$ edge. By Assumption 3, we know that $[\beta] \in \Omega$, for which $[\beta] \in \mathbb{T}_{c'}(i)$ holds by construction of $\textsc{Finish}(G)$. 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 $O_{c'}(i) = 0$.

Algorithm 4 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** INIT(1), 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 λ, τ 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** 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: $s = s \oplus$ APPLY(o_j, a_j)
20: **end for**
21: $s = s \oplus$ FINISH(G) where $\tau_G = \lambda$
22: **return** s
23: **end function**

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

□

Lemma G.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 G.15. Let c be a configuration derived by an LTL transition sequence s that contains j FINISH transitions. Then $C_{LTL}(c)$ (Algorithm 4) generates a valid sequence s' of LTL transitions that leads to a goal configuration c' or $s \oplus s'$ contains $j + 1$ FINISH transitions.

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 G.2, we have to show that $\mathbb{G}_{c'}$ is defined for some token i. There is only one path through Algorithm 4 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 4 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 G.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 G.13.

- Line 19 assumes that $\text{APP}_{o_j} \in L$. This is guaranteed by Assumption 4.

□

We can construct the transition sequence for which Theorem G.9 asks by repeatedly applying C_{LTL} to a given configuration c. Lemma G.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 G.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}.