A Unified Encoding of Structures in Transition Systems

Tao Ji[*], **Yong Jiang, Tao Wang, Zhongqiang Huang,**
Fei Huang, Yuanbin Wu, Xiaoling Wang
School of Computer Science and Technology,
East China Normal University
DAMO Academy, Alibaba Group
`{taoji@stu,ybwu@cs,xlwang@cs}.ecnu.edu.cn`
`{yongjiang.jy,leeo.wangt,z.huang,f.huang}@alibaba-inc.com`

Abstract

Transition systems usually contain various dynamic structures (e.g., stacks, buffers). An ideal transition-based model should encode these structures completely and efficiently. Previous works relying on templates or neural network structures either only encode partial structure information or suffer from computation efficiency. In this paper, we propose a novel attention-based encoder unifying representation of all structures in a transition system. Specifically, we separate two views of items on structures, namely structure-invariant view and structure-dependent view. With the help of parallel-friendly attention network, we are able to encoding transition states with $\mathcal{O}(1)$ additional complexity (with respect to basic feature extractors). Experiments on the PTB and UD show that our proposed method significantly improves the test speed and achieves the best transition-based model, and is comparable to state-of-the-art methods. [1]

1 Introduction

Transition systems have been successfully applied in many fields of NLP, especially parsing (dependency parsing (Nivre, 2008), constituent parsing (Watanabe and Sumita, 2015), and semantic parsing (Yin and Neubig, 2018)). Basically, a transition system takes a series of actions which attach or detach some *items* (e.g., sentence words, intermediate outputs) to or from some *structures* (e.g., stacks, buffers, partial trees). Given a set of action series, a classifier is trained to predict the next action given a current *configuration* of structures in the transition system. The performances of the final system strongly depend on how well the classifier encodes those transition system configurations.

Ideally, a good configuration encoder should encode transition system structures completely and efficiently. However, challenges appear when we try to have the cake and eat it. For example, traditional template-based methods (Chen and Manning, 2014) are fast, but only encode partial information of structures (e.g., few top items on stacks and buffers). Structure-based networks (e.g., Stack-RNN (Dyer et al., 2015)) rely on carefully designed network architecture to get a full encoding of structures (actually, they still miss some *off-structure* information, see our discussions in Section 4.2), but they are usually slow (e.g., not easy to batch). Furthermore, different structures have different ways of update (stacks are first-in-last-serve, buffers are first-in-first-serve), it also takes efforts to design different encoders and ways of fusing those encoders.

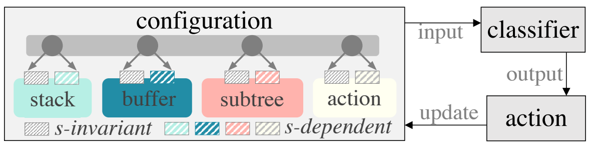


Figure 1: An overview of our transition-based parser.

In this work, we aim to provide a unified encoder for different transition system structures. Instead of inspecting different structures individually, to unify the encoding, we turn to inspect items in each structure, which are ultimate targets for any structure encoder. One key observation is that every item has two-views, namely *structure-invariant view* which is unchanged when the item is placed on different structures, and *structure-dependent view* which reflects which part of which structure the item stands. For example, when a word w (item) is on the buffer (structure), its structure-invariant view could contain its lexical form and part-of-speech tag, while its structure-dependent view indicates that w is now sitting on the buffer and its distance to the buffer head is p. When w is detached from buffer and attached to the stack, its structure-dependent view will switch to "sitting on the stack" while its structure-invariant view stay unchanged. A unified

[*]This work was conducted when Tao Ji was interning at Alibaba DAMO Academy.
[1]https://github.com/AntNLP/trans-dep-parser.

structure encoder thus suffices to uniformly encode both views.

For the structure-invariant view, we share them among different structures, thus it is automatically unified. For the structure-dependent view, we propose a simple yet powerful encoder. It assigns each structure a set of indicating vectors (*structure indicators*), each indicator specifies certain part of that structure. For example, we use indicators (vectors) to expressing "on top of the stack", "the second position of the buffer", and "index of head words in partial trees". To encode an item, we only need to concatenate its structure-invariant encoding and corresponding indicators according to its position in that structure.

Regarding completeness and efficiency, we find that with structure indicators, it is relatively easy to encode a structure completely: one only needs to decompose the structure into identifiable subparts. In fact, we can use them to track some parts of structures which are not revealed in previous work (e.g., words have been popped out from stacks). It runs in the same manner as templated-based models, thus the decoding efficiency is guaranteed. We also note that using structure indicator is different from existing ways to include structure information into neural network models (Shaw et al., 2018; Wang et al., 2019; Shiv and Quirk, 2019): it encodes dynamical structures (changing with transition system running) rather than static structures (e.g., fixed parse trees).

We can easily implement the unified structure encoding with existing multi-head attention networks (MHA, (Vaswani et al., 2017)). It is also easy to fuse encodings of different structures with multi-layer MHA. We conduct experiments on the English Penn Treebank 3.0 and Universal Dependencies v2.2, show that the unified structure encoder is able to help us achieving state-of-the-art transition-based parser (even competitive to the best graph-based parser), while retaining a fast training and testing speed.

2 Transition Systems

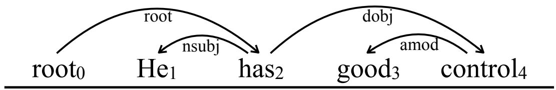
We briefly review transition-based dependency parsing. Given a sentence $x = \text{root}_0, w_1, \cdots, w_n$ (root_0 is a synthetic word) and a relation set $\mathbb{R}$, we denote a dependency tree for x to be $\{(i, j, r)\}$, where (i, j, r) represents a dependency relation $r \in \mathbb{R}$ between w_i (head) and w_j (dependent).

A transition system is a sound quadruple $S = (\mathbb{C}, \mathbb{A}, c_x, \mathbb{C}_t)$, where $\mathbb{C}$ is the set of configurations. $\mathbb{A}$ is the set of actions, c_x is an initialization function mapping x to a unique initial configuration, and $\mathbb{C}_t \subseteq \mathbb{C}$ is a set of terminal configurations. Given a configuration $c \in \mathbb{C}$, a transition-based parser aims to predict a correct action $a \in \mathbb{A}$ and move to a new configuration. We specifically describe the arc-hybrid system (Kuhlmann et al., 2011). In this system, each configuration $c = (\sigma|i, j|\beta, \mathbb{T})$ aggregates information from three structures, namely a stack (σ, where $\sigma|i$ denotes the stack top is w_i), a buffer (β, where $j|\beta$ denotes the buffer front is w_j) and a partial tree ($\mathbb{T}$). The actions of arc-hybrid are formalized as (a running example is shown in Figure 2):

$$\begin{array}{rcll} (\sigma,\ i|\beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}) & (\texttt{sh}) \\ (\sigma|i,\ j|\beta,\ \mathbb{T}) & \vdash & (\sigma,\ j|\beta,\ \mathbb{T} \cup (j, i, r)\ ) & (\texttt{la}_r) \\ (\sigma|i|j,\ \beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T} \cup (i, j, r)\ ) & (\texttt{ra}_r) \end{array}$$

There are three actions, `sh` moves the front item of the buffer (w_i) to the top item of the stack. $\texttt{la}_r$ removes the top item the stack w_i, attaches it as a dependent to w_j with label r, and adds a left-arc (j, i, r) to the partial tree. $\texttt{ra}_r$ removes the top of the stack w_j attaches it as a dependent to w_i with label r, and adds a right-arc (i, j, r) to the partial tree. We note that all actions are actually attaching or detaching item to or from structures, where *an item* could be a word (in stack and buffer) or an edge (in the partial tree).

Note that besides structures in the configurations, we can also incorporate other structures to help learning action predictors. For example, we can consider the history action list which contain all



t	a	Configuration		
		([ ],	$[1, \ldots, 4, 0]$,	$\varnothing$)
1	sh	([1],	$[2, 3, 4, 0]$,	)
2	la	([ ],	$[2, 3, 4, 0]$,	$\cup(2, 1, nsubj)$)
3	sh	([2],	$[3, 4, 0]$,	)
4	sh	([2, 3],	$[4, 0]$,	)
5	la	([2,],	$[4, 0]$,	$\cup(4, 3, amod)$)
6	sh	([2, 4],	$[0]$,	)
7	ra	([2,],	$[0]$,	$\cup(2, 4, dobj)$)
8	la	([ ],	$[0]$,	$\cup(0, 2, root)$)

Figure 2: A running example of the *Arc-hybrid* transition-based parsing. The above gold tree is constructed after performing 8 correct actions. We will use the grey row as an example for the structural indicator.

previous actions in a sequential manner. In this case, an item in this action list is an action label.

3 Two Views of an Item

We can see that, while its "content" remains the same, an item may appear in different structures in a transition system's configurations. To uniformly encode an item (and thus structures containing them), we can decouple the encoding of contents and structures, then combine them in a unified way. This simple method also suggests us to design unified structure encoders which make the whole transition system model concise and efficient.

The *structure-invariant view* typically captures the lexical (shallow) form of an item. For example, in the arc-hybrid system, items in stacks and buffers have words as their structure-invariant view, items in action list have actions as their structure-invariant view. This view is shared when the item moving from one structure to another, and we only need to encode it once (e.g., no matter the stack or the buffer a word appears, its structure-invariant representation is identical). We describe how to encode this view in Section 4.3.

The more interesting problem is how to characterize the *structure-dependent view*. We would like to have a unified strategy to represent those structures. Our major tool is *structure indicators*, which are basically a set of vectors bounded to each structure. Taking the stack for example. We use vectors to indicate "the top of stack" (we name the vector with "1_σ"), "the second to the stack top" (naming with "2_σ"). Vector "0_σ" indicates the parts haven't been in the stack. Different with previous work, we could also represent "the previous stack top which has been popped" (a vector naming by "-1_σ"), and "the previous previous stack top" (a vector naming by "-2_σ"). That is, for different parts of the structure, we employ vectors to indicate them.

Similarly, for the buffer we have another set of structure indicators $\{1_\beta, -1_\beta, 2_\beta, -2_\beta \cdots\}$, where a positive number indicates the position in the buffer, a negative number indicates the time step passed since the item has been removed from the buffer. For the partial tree with dependency relation, we decompose it into two structures, the tree arc ($\mathbb{T}_{arc}$) and the dependency relation ($\mathbb{T}_{rel}$). A set of $\mathbb{T}_{arc}$ indicators $\{0_{\mathbb{T}_{arc}}, 1_{\mathbb{T}_{arc}}, -1_{\mathbb{T}_{arc}} \cdots\}$ indicates the position from item to its head word. A set of $\mathbb{T}_{rel}$ indicators $\{0_{\mathbb{T}_{rel}}, 1_{\mathbb{T}_{rel}}, 2_{\mathbb{T}_{rel}}, \cdots, |\mathbb{R}|_{\mathbb{T}_{rel}}\}$ indicates the IDs of dependency relations. Vectors "$0_{\mathbb{T}_{arc}}$" and "$0_{\mathbb{T}_{rel}}$" indicate that this dependency edge is not in partial tree. For the action list we have a set of structure indicators $\{1_a, 2_a, \cdots\}$ where a vector incicates the position in the list.

	root$_0$	He$_1$	has$_2$	good$_3$	control$_4$
σ	0	-1	2	1	0
β	2	-4	-2	-1	1
$\mathbb{T}_{arc}$	0	1	0	0	0
$\mathbb{T}_{rel}$	0	1	0	0	0

	sh	la$_{nsubj}$	sh	sh
α	4	3	2	1

Figure 3: An instance of structure indicators after the 4th step in Figure 2. Grey rows indicate structure-invariant parts ($\sigma, \beta, \mathbb{T}_{arc}$ and $\mathbb{T}_{rel}$ are shared), and other rows indicate structure-dependent parts. To simplify, we express the relation *nsubj* by vector $1_{\mathbb{T}_{rel}}$.

In Figure 3, we show the two-views of an instance from the 4th step in Figure 2. We can observe that the five different structures mentioned above have a unified form now.

4 The Unified Structure Encoder

4.1 Encoding with USE

When a transition system is running at moment t, the parser needs to capture as much information as possible about the current configuration to determine which is the correct action. The key is to encode the configuration containing many structures concisely and efficiently. We propose a unified structure encoder (USE) by using multi-head self-attention networks (Vaswani et al., 2017). Each head extracts a feature vector of one structure (e.g., $\boldsymbol{o}_\sigma$ for the stack).

A common USE function maps a query and a set of key-value pairs to an output. The query vector $\boldsymbol{q}$ represents the current time step and data structure. The key-value pairs both represent two-views of the structure. The output vector $\boldsymbol{o}$ is calculated as a weighted sum of values, where the weight assigned to each value is calculated by a scaled ($\frac{1}{\sqrt{d_k}}$) dot-product function of the query with the corresponding key. In practice, we pack the keys and values into matrices K and V, then compute the output as:

$$\boldsymbol{o} = \mathrm{USE}(\boldsymbol{q}, K, V) = \mathrm{softmax}\left(\frac{\boldsymbol{q} \cdot K^\top}{\sqrt{d_k}}\right) \cdot V \quad (1)$$

It is universal for different structures. Take the stack σ for example, we calculate the feature vector

$\boldsymbol{o}_{\sigma,t}$ by assigning the $\boldsymbol{q}_{\sigma,t}$, $K_{\sigma,t}$, and $V_{\sigma,t}$:

$$\begin{aligned} \boldsymbol{q}_{\sigma,t} &= W_{\sigma}^{Q} \cdot (\boldsymbol{m}_t \oplus \boldsymbol{m}_{\sigma}) \\ K_{\sigma,t} &= W_{\sigma}^{K} \cdot (X + S_{\sigma,t}^{K}) \\ V_{\sigma,t} &= W_{\sigma}^{V} \cdot (X + S_{\sigma,t}^{V}). \end{aligned} \tag{2}$$

Where $\boldsymbol{m}_t$ and $\boldsymbol{m}_{\sigma}$ are the marker embeddings of time step t and data structure σ; W_{σ}^{Q}, W_{σ}^{K}, W_{σ}^{V} are parameter matrices for linear transformation; X is the word embedding matrix [2]. We describe X and A in detail later. The $S_{\sigma,t}^{K}$ and $S_{\sigma,t}^{V}$ are the embedding matrices of the structural indicator. Take σ in Figure 3 for example, $S_{\sigma,t}^{K} = [0_{\sigma}, -1_{\sigma}, 2_{\sigma}, 1_{\sigma}, 0_{\sigma}]^{K}$ and $S_{\sigma,t}^{V} = [0_{\sigma}, -1_{\sigma}, 2_{\sigma}, 1_{\sigma}, 0_{\sigma}]^{V}$. Following Shaw et al. (2018), we use this two sets of structural embeddings for key-value pairs and add them to X to combine the information.

When the system comes to the next moment $t{+}1$, we use the $\boldsymbol{m}_{t+1}$, $S_{\sigma,t+1}^{K}$ and $S_{\sigma,t+1}^{V}$ for an updated configuration. For the other four structures, we calculate their feature vectors $\boldsymbol{o}_{\beta,t}$, $\boldsymbol{o}_{\alpha,t}$, $\boldsymbol{o}_{\mathbb{T}_{arc},t}$, and $\boldsymbol{o}_{\mathbb{T}_{rel},t}$ by assigning the corresponding $\boldsymbol{q}$, K, and V, respectively [3].

4.2 Fusion of Structure Encodings

After obtaining feature vector of each structure, the encoder incorporates all of them into configuration representation $\boldsymbol{c}_t$. Here, we simply use a multi-layer perceptron (MLP):

$$\boldsymbol{c}_t = \text{MLP}(\boldsymbol{o}_{\sigma,t} \oplus \boldsymbol{o}_{\beta,t} \oplus \boldsymbol{o}_{\alpha,t} \oplus \boldsymbol{o}_{\mathbb{T}_{arc},t} \oplus \boldsymbol{o}_{\mathbb{T}_{rel},t}).$$

Besides that, to enhance more interaction among structures, we stack L USE layers and add the previous layer's configuration vector $\boldsymbol{c}_t^{(l-1)}$ $(1 < l \leq L)$ when computing the query vector $\boldsymbol{q}_{*,t}^{(l)}$ ($*$ for any structures).

$$\boldsymbol{q}_{*,t}^{(l)} = W_{*}^{Q(l)} \cdot \left(\boldsymbol{m}_t^{(l)} \oplus \boldsymbol{m}_{*}^{(l)} \oplus \boldsymbol{c}_t^{(l-1)}\right)$$

Since $\boldsymbol{c}_t^{(l-1)}$ contains the complete structural information, the lth layer's USE module can interact with other structures and output a more informative representation $\boldsymbol{o}_{*}^{(l)} = \text{USE}(\boldsymbol{q}_{*}^{(l)}, K_{*}^{(l)}, V_{*}^{(l)})$. Then, we obtain a high layer configuration representation by combining these output vectors:

$$\boldsymbol{c}_t^{(l)} = \text{MLP}(\boldsymbol{o}_{\sigma,t}^{(l)} \oplus \boldsymbol{o}_{\beta,t}^{(l)} \oplus \boldsymbol{o}_{\alpha,t}^{(l)} \oplus \boldsymbol{o}_{\mathbb{T}_{arc},t}^{(l)} \oplus \boldsymbol{o}_{\mathbb{T}_{rel},t}^{(l)}).$$

[2]Note that we use action embedding matrix A instead of X when encoding action list α.

[3]Similar to Equation 2, we give the formulation for the other data structures in Appendix B.

	σ		β		a	$\mathbb{T}$	GPU
	in	out	in	out			☺
Top-k	▲	-	▲	-	-	▲	😄
σ–LSTM	★	-	★	-	★	-	☹
Binary	▲	▲	▲	▲	-	-	😄
USE	★	★	★	★	★	★	😄

Figure 4: Structural information coverage and GPU-friendliness of different feature extractors. ★ indicates complete extraction, ▲ indicates partial extraction, ☹ indicates GPU parallel friendly, and 😄 indicates unfriendly.

We set different layers with different parameters (preliminary experiments suggest shared parameter performs worse). To support deeper networks, the residual connection and layer normalization (Ba et al., 2016) are employed on MLP and USE modules. Finally, we use $\boldsymbol{c}_t^{(L)}$ of the last layer to classify action.

Basically, we need at least 5 attention heads to extract full structures (each head corresponds to one structure). Vaswani et al. (2017) noted that a multi-head attention layer has a constant number ($\mathcal{O}(1)$) of sequentially executed operations, which means that efficient GPU-based computing is possible. In training, the USE calculations at different moments are independent of each other, so we can pack them into the batch dimension to obtain an $\mathcal{O}(1)$ training complexity. Hence, USE can uniformly extract full structure features efficiently.

Comparing to Previous Encoders We divide previous work into three encoding methods: top-k, stack-LSTM, and binary vector. **Top-k** methods (Chen and Manning, 2014; Weiss et al., 2015) capture the conjunction of only few $1{\sim}3$ in-structure items. It extracts only partial structural information. Since the feature template is fixed, it is easy to batchify. **Stack-LSTM** methods (Dyer et al., 2015; Ballesteros et al., 2016) can efficiently represent all in-structure items, via the `PUSH`$(\cdot)$ and `POP`$(\cdot)$ functions. But it loses the information of outside parts and subtree which cannot be treated as a stack. Besides, Che et al. (2019) point out that its batch computation is very inefficient. **Binary Vector** methods (Zhang et al., 2017) use two binary vectors to model whether each element is in a σ or a β. It can efficiently encode some outside parts of stack and buffer but loss the information of inside position.

We compare existing work with our USE encoder in terms of the coverage of structure features and GPU computing friendly (in Figure 4). Overall, USE does not lose any structural information and more efficient than previous feature extraction schemes.

4.3 Encoding the Structure-invariant View

Given a sentence $s = \omega_0, \ldots, \omega_n$, we learn a lexical vector $\boldsymbol{x}_i$ for each word ω_i, and pack them into matrix X for Equation 1. The vector $\boldsymbol{x}_i$ is composed of three parts: the word embedding $\boldsymbol{e}(\omega_i)$, the part-of-speech (POS) tag embedding $\boldsymbol{e}(g_i)$, and the character-level representation vector $\text{CharCNN}(\omega_i)$.

$$\boldsymbol{x}_i = \boldsymbol{e}(\omega_i) \oplus \boldsymbol{e}(g_i) \oplus \text{CharCNN}(\omega_i). \quad (3)$$

We simply initialize all embedding matrices in a random way. The $\text{CharCNN}(\omega_i)$ vector is obtained by feeding ω_i into a character convolutional neural network (Zhang et al., 2015). To encode more sentence context, $\boldsymbol{e}(\omega_i)$ is obtained by trainable bidirectional long short-term memory, Transformer encoder networks or pre-trained networks like Bert.

Given an action list $\alpha = a_0, \ldots, a_m$, we learn a structure-invariant vector $\boldsymbol{a}_i$ for each action a_i. Because the action space is only $2|\mathbb{R}| + 1 (< 10^2)$, we directly obtain $\boldsymbol{a}_i = \boldsymbol{e}(a_i)$ by action embedding.

Since all decoding steps share the same structure-invariant representations, they are just computed only once. In the experiments, we will discuss all mentioned encoding ways.

5 The Action Classifier

The action set $\mathbb{A}$ is first divided into three main types: `sh`, `la` and `ra`, then divided into $|\mathbb{R}|$ dependency labels only for `la` and `ra` actions. Thus we perform a two-stage process with 3-class and $|\mathbb{R}|$-class classifications. It effectively reduces the classification space compared with one-stage process. For example, the space of a `sh` action is 3-class in two-stage process while $2|\mathbb{R}|+1$ class in one-stage process.

For the action type classification, based on $\boldsymbol{c}_t^{(L)}$, we follow Kiperwasser and Goldberg (2016) which scores the three actions by an MLP,

$$\text{Score}\left(t, \begin{bmatrix}\texttt{sh}\\\texttt{la}\\\texttt{ra}\end{bmatrix}\right) = \text{MLP}\left(\boldsymbol{c}_t^{(L)}\right)\begin{bmatrix}\texttt{sh}\\\texttt{la}\\\texttt{ra}\end{bmatrix}. \quad (4)$$

To classify the dependency label r between word i and j, based on the lexical representations $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ and $\boldsymbol{c}_t^{(L)}$, we follow Dozat and Manning (2017) which uses a biaffine score function to predict the label's probability,

$$\begin{aligned}\boldsymbol{z} &= \boldsymbol{x}_i^\top \boldsymbol{W}_1 \boldsymbol{x}_j + \left(\boldsymbol{x}_i \oplus \boldsymbol{x}_j \oplus \boldsymbol{c}_t^{(L)}\right)^\top W_2 + \boldsymbol{b}\\ P(r|i,j) &= \text{Softmax}(\boldsymbol{z})[r]. \quad (5)\end{aligned}$$

Where $\boldsymbol{W}_1$ is a 3-dimensional parameter tensor, W_2 is a parameter matrix, and $\boldsymbol{b}$ is a parameter vector [4]. A slight difference is that we induce $\boldsymbol{c}_t^{(L)}$ to model the prior probability of each label under the current configuration.

5.1 Training Details

We have two training objectives, one is scoring the correct action higher than the incorrect action, and the second one is maximizing the probability of the correct dependency label. For correct action α^*, aiming to maximize the margin between its score and the highest incorrect action ($\hat{\alpha}$) score, we use the *hinge loss*:

$$\begin{aligned}\mathcal{L}_\alpha = \frac{1}{2n}\sum_{t=1}^{2n} \max\big(0, 1 - &\text{Score}(t, \alpha^*)\\ &+ \max_{\hat{\alpha}\neq\alpha^*} \text{Score}(t, \hat{\alpha})\big).\end{aligned}$$

For correct dependency label r^*, aiming to maximize its probability, we use the *cross-entropy loss*:

$$\mathcal{L}_r = \frac{1}{n}\sum_{(i,j,r^*)\in\mathbb{T}} -\log P(r^*|i,j).$$

The final objective is to minimize a weighted combination of them: $\mathcal{L} = \lambda_1\mathcal{L}_\alpha + \lambda_2\mathcal{L}_r$.

We follow Kiperwasser and Goldberg (2016) which use error exploration training with dynamic-oracle and aggressive strategies. A parser that always takes the correct action during training will suffer from error propagation during testing. To take wrong actions, the dynamic-oracle well-defines the “correct” actions even if the current configuration cannot lead to the gold tree. The aggressive exploration forces taking a wrong action with probability $p_{agg} = 0.1$.

6 Interprete the Features

Since a transition system contains various structures, a natural question is *which part of structures*

[4]Note that in cases of Bert representation, we reduce the dimension of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ by linear transformation before computing Equation 5.

are important for the parser? To show the importance of a variable, one standard approach is using partial derivatives multiplied by the variable's value (Denil et al., 2015). Hence, the importance score ($\triangle$) of a structural indicator is a dot product between objective function gradient and indicator embedding:

$$\triangle(\sigma, i) = \left(\nabla_{\boldsymbol{s}^K_{\sigma,i}} \mathcal{L}\right)^\top \cdot \boldsymbol{s}^K_{\sigma,i} + \left(\nabla_{\boldsymbol{s}^V_{\sigma,i}} \mathcal{L}\right)^\top \cdot \boldsymbol{s}^V_{\sigma,i}$$

Concretely, $\triangle(\sigma, i)$ shows the relevance between the stack indicator i and the decision of our parser. The importance at indicator i of any structure can be derived similarly. We can further accumulate multiple items' relevance. For example, stack inside relevance $\triangle_{in}(\sigma) = \sum_{i>0} \triangle(\sigma, i)$, stack outside relevance $\triangle_{out}(\sigma) = \sum_{i\leq 0} \triangle(\sigma, i)$, etc. In the experiments, we explain the importance of each structure part by $\triangle$ score.

7 Experiments

Data We conduct experiments and analysis on two main datasets including 12 languages: the English Penn Treebank (PTB 3.0) with Stanford dependencies, and the Universal Dependencies (UD 2.2) (Nivre et al., 2018) treebanks used in CoNLL 2018 shared task (Zeman et al., 2018). The statistics of datasets are in Appendix C. For PTB, we use the standard train/dev/test splits and the external POS tags obtained by the Stanford tagger (accuracy $\approx 97.3\%$). Following Ji et al. (2019), we select 12 languages from UD, and use CoNLL shared task's official train/dev/test splits, where the POS tags were assigned by the UDPipe (Straka et al., 2016).

Evaluation We mainly report unlabeled (UAS) and labeled attachment scores (LAS). For evaluations on PTB, five punctuation symbols (“ ” : , .) are excluded, while on UD, we use the official evaluation script.

Hyper-parameters For *structure-invariant* part, we directly adopt most parameter settings of Ji et al. (2019) and Zhang et al. (2020), including pretrained embeddings, BiLSTM, and CharCNN. For *structure-dependent* part, we use a total of 8 structural heads, allocating two each for the stack, buffer and action list, one for the subtree's edges and one for the edges' labels. Our pre-experiments show that stacking 6 layers of **USE** yields the best results. The weight λ of the objective function is assigned to 0.5. We trained our parser for up to 1k iterations, stopping early if peak performance on dev did not increase over 100 epochs. The details of the chosen hyper-parameters in default settings are summarized in Appendix D.

Parser	Type	Test UAS	Test LAS
Chen and Manning (2014)	T	91.8	89.6
Weiss et al. (2015)	T	94.26	91.42
Andor et al. (2016)	T	94.61	92.79
Dyer et al. (2015)	T	93.1	90.9
Ballesteros et al. (2016)	T	93.56	92.41
Kiperwasser and Goldberg (2016)	T	93.1	91.0
Zhang et al. (2017)	T	93.71	91.60
Mohammadshahi and Henderson (2020)	T	93.07	91.08
Ma et al. (2018)	T	95.87	94.19
Yuan et al. (2019)	T	94.60	94.02
Dozat and Manning (2017)	G	95.74	94.08
Li et al. (2019)	G	95.93	94.19
Ji et al. (2019)	G	95.97	94.31
Zhang et al. (2020)	G	96.14	94.49
Our **USE** Parser			
arc-hybrid	T	**95.99**	**94.28**
arc-standard	T	95.95	94.26
arc-eager	T	95.93	94.23

Table 1: Results on the English PTB dataset. “T” represents transition-based parsers, and “G” represents graph-based parsers. We report the average over 5 runs.

7.1 Main Results

Firstly, we compare our method with previous work (Table 1). The first part contains transition-based models. We particularly compare with the two strong baselines in the blue cell, where Ma et al. (2018) decode parse trees in a depth-first manner with a stack-pointer network, and Yuan et al. (2019) decode transition sequences in both the forward and backward directions by multi-task learning. In a fair comparison, our three unified structure encoding (**USE**) parsers all achieve significant improvements on PTB. This demonstrates the benefit of complete structural information by our unified encoding.

Secondly, we compare with strong graph-based parsers. The second part of Table 1 contains two first-order parsers and two high-order parsers (in the red cell). Our **USE** parsers beat the first-order methods, but underperform the high-order methods which capture high-order features by graph neural networks and TreeCRF. However, speed experiments show that **USE** is about 2 times faster than them, It's our future work to bridge the performance gap by using the bi-directional transition system (Yuan et al., 2019) and stronger decoding

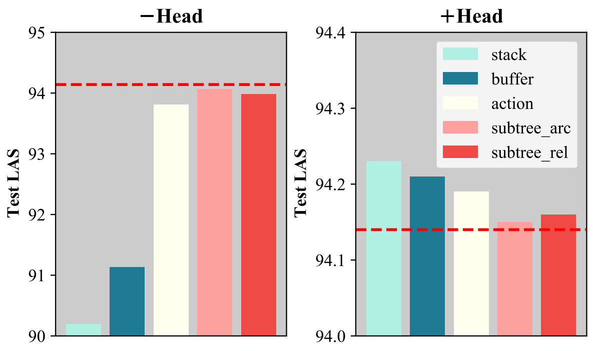


Figure 5: Analysis of the allocation of structural heads. The red line is the performance of basic setup.

methods (Andor et al., 2016).

Thirdly, we compare the results of three **USE** parsers with different transition systems (third part of Table 1). We can see that the *arc-hybrid* system is more expressive than the *arc-eager* and the *arc-standard*. Shi et al. (2017) demonstrate that the *arc-eager* is more expressive on a minimal feature set, but our results do not support them on a full feature set. The reason may be that, when the feature set is full, *arc-eager* system has one more action (REDUCE) than *arc-standard* in the first stage of classification.

Head Allocation Here we discuss the allocation of structural heads. Our basic idea is assigning one head to one structure, which means five heads in total. We performe two sets of ablation experiments based on the basic setup: decreasing or increasing one head for each structure respectively (Figure 5). Decreasing one head means that the corresponding structure is not visible to the parser. Losing the information of stack or buffer severely hurts the performance. Comparatively, losing the information of action list or subtree slightly hurts the performance. This suggests that the stack and buffer are more important in *arc-hybrid* transition system, and we should pay more attention to them. Increasing one head shows the improved performance of giving the corresponding structure double attention. We observe obvious performance gains on the stack, buffer, and action list, which means that augmenting their information is helpful. Considering the performance gain and computational cost of adding heads, we finally use a total of 8 structural heads. The parser double attent to the stack, buffer and action list.

Lexical Representation We analyze different lexical word representation from Section 4.3 (Table 2). The first part reports the use of context-

Lexical Encoder	**Dev**		**Test**	
	UAS	LAS	UAS	LAS
Glove	95.72	93.79	95.71	94.05
+ BiLSTM	95.81	93.87	95.93	94.21
+ Xformer	**95.84**	**93.93**	**95.99**	**94.28**
Bert	95.90	93.97	96.21	94.56
+ finetune	**95.97**	**94.02**	**96.28**	**94.60**
M&H20	95.78	93.74	96.11	94.33

Table 2: Lexical encoder comparison on PTB. M&H20: Mohammadshahi and Henderson (2020).

Parser	Type	**Speed**
Ma et al. (2018)	T	183
Dozat and Manning (2017)	G	496
Ji et al. (2019)	G‡	403
Zhang et al. (2020)	G‡	466
Our *arc-hybrid* parser	T	918

Table 3: Parsing speed comparison on PTB test set. The ‡ indicates high-order graph-based parsers.

independent Glove embeddings (Pennington et al., 2014) in the *arc-hybrid* system. We learn the context via BiLSTM or Transformer encoder. The results show that encoding context can further improve performance and the Transformer encoder is better than BiLSTM. The second part reports the use of contextual Bert networks (Devlin et al., 2019). The introduction of Bert networks and in particular fine-tuning usage can significantly increase the performance. Compared with Mohammadshahi and Henderson (2020), our parser performs better because it encodes the full structure rather than only top-k in-structure items.

Parsing Speed Table 3 compares the parsing speed of different parsers on PTB test set. For a fair comparison, we run all parsers with python implementation on the same machine with Intel Xeon E5-2650v4 CPU and GeForce GTX1080Ti GPU. The **USE** parser can parse about 918 sentences per second, over 5 times faster than the strongest transition-based parser (Ma et al., 2018). This result shows the efficiency of the attention mechanism. Compared to three graph-based parsers, our parser is nearly 2 times faster than theirs. It's because the transition-based parser decodes linearly and does not require complex decoding algorithms like minimum spanning tree or TreeCRF. Considering the parsing performance and speed together, our proposed parser is able to meet the requirements of a real-time system.

	bg	ca	cs	de	en	es	fr	it	nl	no	ro	ru	Avg.
Ma18	89.31	90.55	89.62	77.75	82.32	90.28	85.83	90.75	87.57	89.82	85.34	92.06	87.60
Zhang20	89.72	**91.27**	**90.94**	78.26	**82.88**	**90.79**	**86.33**	**91.02**	87.92	**90.17**	85.71	**92.49**	**88.13**
arc-hybrid	**89.81**	90.91	90.68	**78.48**	82.52	90.27	85.98	90.83	**87.96**	89.91	**85.88**	92.36	87.97

Table 4: LAS on UD2.2 test datasets. Ma18: Ma et al. (2018); Zhang20: Zhang et al. (2020). We report the average over 3 runs.

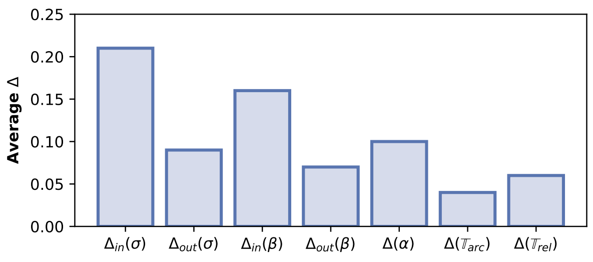


Figure 6: Analysis of the importance score (△) for different structure part.

Interpretability Figure 6 visualizes the importance score (△) of each structure part in *arc-hybrid* transition system. Consistent with the findings in Figure 5, the stack and buffer achieve higher importance scores. However, in reaching the same conclusion, the interpretable method does not require retraining of the parser. Furthermore, we observe that the outside information of the stack and buffer is more important than the subtree structure. It suggests that the transition parser should encode them.

UD Treebanks Table 4 compares our **USE** parser with two baselines on UD datasets. We adopt the non-projective *arc-hybrid* system for handling the ubiquitous non-projective trees (de Lhoneux et al., 2017). As the transition-based baseline, the parser proposed by Ma et al. (2018) was re-runned under the same hyper-parameter settings as ours. Our **USE** parser outperforms the baseline on all of the 12 languages, the averaged improvement is 0.37 LAS, again showing the power of the complete transition system encoder. Compared with the strongest graph-based baseline (Zhang20), our parser performs better on 4 treebanks, including bg, de, nl, and ro. These four treebanks are relatively smaller than other treebanks, probably indicating that our parser is more suitable for low resource languages. Overall, there is still a 0.15 averaged LAS gap with the graph-based baseline, and it is our future work to further improve the **USE** transition-based parser.

8 Related Work

We have already surveyed related transition system encoder in Section 4.2. Here we present several powerful transition-based parsers. Ma et al. (2018) decode a parse tree step-by-step based on a depth-first traversal order. A stack is usually used to maintain the depth-first search. Thus they use a stack-pointer network for decoding. Note that their work is not based on any transition systems. Yuan et al. (2019) propose a bidirectional decoding method for a stack-LSTMs transition-based parser. They perform joint decoding with a left-to-right parser and a right-to-left parser. Mohammadshahi and Henderson (2020) propose a Graph2Graph framework for enhancing expression by treating multiple structures as multiple sentences and using a Transformer encoder (Bert) to encode top-k words. These works focus on improving the decoding approach or representation learning of structure-invariant parts, but still follow the traditional encoders. Our work focuses on proposing a new encoder with both information completeness and computational effectiveness.

There have been several attempts to combine attention networks with structures: to represent the sequential structure better, Shaw et al. (2018) introduce relative position between words in attention networks instead of concatenating absolute position in input. Wang et al. (2019) define the relative positions on parse trees to encode each word pair's tree distance. They feed these positional embeddings to attention networks too. These two works encode a static structure, while we encode a dynamically changing transition system. Shiv and Quirk (2019) extend the Transformer's sinusoidal position function to the tree structure. Similar to us, their decoder dynamically computes the new position encoding when generating a tree structure. But their structural embeddings are computed by fixed sinusoidal function, while ours are learnable. These works encode only one structure, while we encode multiple structures from a transition system.

9 Conclusion

We presented a comprehensive and efficient encoder for transition system. We separate each structure to the structure-invariant part and structure-dependent part. It allows us to dynamically encode the complete structure and also retains the efficiency of training and testing. Experiments show that the proposed parser achieves new state-of-the-art transition-based results.

Acknowledgments

The authors wish to thank the reviewers for their helpful comments and suggestions, thank Peng Li and Zhengyi Lei for their comments on writing. This research is funded by the NSFC (62076097) and the 2020 East China Normal University Future Scientists and Outstanding Scholars Incubation Programme (WLKXJ2020). The corresponding authors are Tao Ji, Yuanbin Wu and Xiaoling Wang.

References

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *CoRR*, abs/1607.06450.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A. Smith. 2016. Training with exploration improves a greedy stack LSTM parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2005–2010.

Wanxiang Che, Longxu Dou, Yang Xu, Yuxuan Wang, Yijia Liu, and Ting Liu. 2019. HIT-SCIR at MRP 2019: A unified pipeline for meaning representation parsing via efficient training and effective encoding. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning, CoNLL 2019, Hong Kong, Novemer 3, 2019*, pages 76–85. Association for Computational Linguistics.

Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 740–750.

Miryam de Lhoneux, Sara Stymne, and Joakim Nivre. 2017. Arc-hybrid non-projective dependency parsing with a static-dynamic oracle. In *Proceedings of the 15th International Conference on Parsing Technologies*, pages 99–104, Pisa, Italy. Association for Computational Linguistics.

Misha Denil, Alban Demiraj, and Nando de Freitas. 2015. Extraction of salient sentences from labelled documents. *CoRR*, abs/1412.6815v2.

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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 334–343.

Tao Ji, Yuanbin Wu, and Man Lan. 2019. Graph-based dependency parsing with graph neural networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2475–2485, Florence, Italy. Association for Computational Linguistics.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*, 4:313–327.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA. Association for Computational Linguistics.

Ying Li, Zhenghua Li, Min Zhang, Rui Wang, Sheng Li, and Luo Si. 2019. Self-attentive biaffine dependency parsing. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 5067–5073. ijcai.org.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. 2018. Stack-pointer networks for dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1403–1414, Melbourne, Australia. Association for Computational Linguistics.

Alireza Mohammadshahi and James Henderson. 2020. Graph-to-graph transformer for transition-based dependency parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 3278–3289. Association for Computational Linguistics.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*, pages 149–160, Nancy, France.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain. Association for Computational Linguistics.

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

Joakim Nivre et al. 2018. Universal Dependencies 2.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/1-1983xxx`.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, New Orleans, Louisiana. Association for Computational Linguistics.

Tianze Shi, Liang Huang, and Lillian Lee. 2017. Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 12–23, Copenhagen, Denmark. Association for Computational Linguistics.

Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 12058–12068.

Milan Straka, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*, Portorož, Slovenia. European Language Resources Association.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Xing Wang, Zhaopeng Tu, Longyue Wang, and Shuming Shi. 2019. Self-attention with structural position representations. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 1403–1409. Association for Computational Linguistics.

Taro Watanabe and Eiichiro Sumita. 2015. Transition-based neural constituent parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1169–1179, Beijing, China. Association for Computational Linguistics.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 323–333.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pages 7–12. Association for Computational Linguistics.

Yunzhe Yuan, Yong Jiang, and Kewei Tu. 2019. Bidirectional transition-based dependency parsing. In

The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pages 7434–7441. AAAI Press.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–20, Brussels, Belgium. Association for Computational Linguistics.

Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657.

Yu Zhang, Zhenghua Li, and Min Zhang. 2020. Efficient second-order TreeCRF for neural dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3295–3305, Online. Association for Computational Linguistics.

Zhirui Zhang, Shujie Liu, Mu Li, Ming Zhou, and Enhong Chen. 2017. Stack-based multi-layer attention for transition-based dependency parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1677–1682. Association for Computational Linguistics.

Supplementary Material for
A Unified Encoding of Structures in Transition Systems

A Other Transition Systems

Similar to Section 2, the *arc-standard* system from Nivre (2004) is formally defined as:

$$\begin{array}{rcll} & & ([\text{root}_0],\ [1,\ldots,n],\ \varnothing) & (\mathtt{c}_x) \\ & & ([\text{root}_0],\ [\ ],\ \mathbb{T}_t) & (\mathbb{C}_t) \\ (\sigma,\ i|\beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}) & (\mathtt{sh}) \\ (\sigma|i|j,\ \beta,\ \mathbb{T}) & \vdash & (\sigma|j,\ \beta,\ \mathbb{T}\cup(j,i,r)\ ) & (\mathtt{la}_r) \\ (\sigma|i|j,\ \beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}\cup(i,j,r)\ ) & (\mathtt{ra}_r) \end{array}$$

The *arc-eager* system from Nivre (2003) is formally defined as:

$$\begin{array}{rcll} & & ([\ ],\ [1,\ldots,n],\ \varnothing) & (\mathtt{c}_x) \\ & & (\sigma,\ [\ ],\ \mathbb{T}_t) & (\mathbb{C}_t) \\ (\sigma,\ i|\beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}) & (\mathtt{sh}) \\ (\sigma|i,\ \beta,\ \mathbb{T}) & \vdash & (\sigma,\ \beta,\ \mathbb{T}) & (\mathtt{rd}) \\ (\sigma|i,\ j|\beta,\ \mathbb{T}) & \vdash & (\sigma,\ j|\beta,\ \mathbb{T}\cup(j,i,r)\ ) & (\mathtt{la}_r) \\ (\sigma|i,\ j|\beta,\ \mathbb{T}) & \vdash & (\sigma|i|j,\ \beta,\ \mathbb{T}\cup(i,j,r)\ ) & (\mathtt{ra}_r) \end{array}$$

The non-projective *arc-hybrid* from de Lhoneux et al. (2017) for UD treebanks is formally defined as:

$$\begin{array}{rcll} & & ([\ ],\ [1,\ldots,n,\text{root}_0],\ \varnothing) & (\mathtt{c}_x) \\ & & ([\ ],\ [\text{root}_0],\ \mathbb{T}_t) & (\mathbb{C}_t) \\ (\sigma,\ i|\beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}) & (\mathtt{sh}) \\ (\sigma|i,\ j|\beta,\ \mathbb{T}) & \vdash & (\sigma,\ j|i|\beta,\ \mathbb{T}) & (\mathtt{swap}) \\ (\sigma|i,\ j|\beta,\ \mathbb{T}) & \vdash & (\sigma,\ j|\beta,\ \mathbb{T}\cup(j,i,r)\ ) & (\mathtt{la}_r) \\ (\sigma|i|j,\ \beta,\ \mathbb{T}) & \vdash & (\sigma|i,\ \beta,\ \mathbb{T}\cup(i,j,r)\ ) & (\mathtt{ra}_r) \end{array}$$

B USE for Other Structures

Here we give the formal definitions ($\boldsymbol{q}_{*,t}$, $K_{*,t}$, $V_{*,t}$ in Equation 1) of buffer β, action list α, subtree's arcs $\mathbb{T}_{(arc)}$, and subtree's relations $\mathbb{T}_{(rel)}$, which are omitted in the main text.

$$\begin{aligned} \boldsymbol{q}_{\beta,t} &= W^Q_\beta \cdot (\boldsymbol{m}_t \oplus \boldsymbol{m}_\beta) \\ K_{\beta,t} &= W^K_\beta \cdot (X + S^K_{\beta,t}) \\ V_{\beta,t} &= W^V_\beta \cdot (X + S^V_{\beta,t}). \end{aligned} \quad (6)$$

$$\begin{aligned} \boldsymbol{q}_{\alpha,t} &= W^Q_\alpha \cdot (\boldsymbol{m}_t \oplus \boldsymbol{m}_\alpha) \\ K_{\alpha,t} &= W^K_\alpha \cdot (A + S^K_{\alpha,t}) \\ V_{\alpha,t} &= W^V_\alpha \cdot (A + S^V_{\alpha,t}). \end{aligned} \quad (7)$$

$$\begin{aligned} \boldsymbol{q}_{\mathbb{T}_{(arc)},t} &= W^Q_{\mathbb{T}_{(arc)}} \cdot (\boldsymbol{m}_t \oplus \boldsymbol{m}_{\mathbb{T}_{(arc)}}) \\ K_{\mathbb{T}_{(arc)},t} &= W^K_{\mathbb{T}_{(arc)}} \cdot (X + S^K_{\mathbb{T}_{(arc)},t}) \\ V_{\mathbb{T}_{(arc)},t} &= W^V_{\mathbb{T}_{(arc)}} \cdot (X + S^V_{\mathbb{T}_{(arc)},t}). \end{aligned} \quad (8)$$

$$\begin{aligned} \boldsymbol{q}_{\mathbb{T}_{(rel)},t} &= W^Q_{\mathbb{T}_{(rel)}} \cdot (\boldsymbol{m}_t \oplus \boldsymbol{m}_{\mathbb{T}_{(rel)}}) \\ K_{\mathbb{T}_{(rel)},t} &= W^K_{\mathbb{T}_{(rel)}} \cdot (X + S^K_{\mathbb{T}_{(rel)},t}) \\ V_{\mathbb{T}_{(rel)},t} &= W^V_{\mathbb{T}_{(rel)}} \cdot (X + S^V_{\mathbb{T}_{(rel)},t}). \end{aligned} \quad (9)$$

C Details of Datasets

The statistics (number of sentences) of the English Penn Treebank (PTB) and Universal Dependency (UD) treebanks are summarized in Table 5 and Table 6 respectively.

	#train	#dev	#test
PTB	39832	1700	2416

Table 5: Statistics of the PTB dataset we used.

Treebanks	#train	#dev	#test
Bulgarian	8907	1115	1116
Catalan	13123	1709	1846
Czech	102993	11311	12203
Dutch	18310	1518	1396
English	12543	2002	2077
French	14554	1478	416
German	13841	799	977
Italian	12838	564	482
Norwegian	29870	4300	3450
Romanian	8043	752	729
Russian	48814	6584	6491
Spanish	28492	4300	2174

Table 6: Statistics of the UD dataset we used.

D Hyper-parameters

The hyper-parameters we used in default settings (Table 7).

	Stack					**Buffer**					**Subtree [arc]**					**Subtree [rel]**					**Action**								
t	*root*	*He*	*has*	*good*	*con trol*	*root*	*He*	*has*	*good*	*con trol*	*root*	*He*	*has*	*good*	*con trol*	*root*	*He*	*has*	*good*	*con trol*	*<s>*	*sh*	*la_n subj*	*sh*	*sh*	*la_a mod*	*sh*	*ra_d boj*	*la_r oot*
0	0	0	0	0	0	5	1	2	3	4	0	0	0	0	0	0	0	0	0	0	1								
1	0	1	0	0	0	4	-1	1	2	3	0	0	0	0	0	0	0	0	0	0	2	1							
2	0	-1	0	0	0	4	-2	1	2	3	0	1	0	0	0	0	1	0	0	0	3	2	1						
3	0	-2	1	0	0	3	-3	-1	1	2	0	1	0	0	0	0	1	0	0	0	4	3	2	1					
4	0	-3	2	1	0	2	-4	-2	-1	1	0	0	0	0	0	0	1	0	0	0	5	4	3	2	1				
5	0	-4	1	-1	0	2	-5	-3	-2	1	0	1	0	1	0	0	1	0	2	0	6	5	4	3	2	1			
6	0	-5	2	-2	1	1	-6	-4	-3	-1	0	1	0	1	0	0	1	0	2	0	7	6	5	4	3	2	1		
7	0	-6	1	-3	-1	1	-7	-5	-4	-2	0	1	0	1	-2	0	1	0	2	3	8	7	6	5	4	3	2	1	

Figure 7: The structured-dependent information for all steps on the example. For simplicity, we use the number 0,1,2,3 to denote the *none*,*nsubj*,*amod*,*dobj* relations in the "subtree[rel]", respectively.

Layer	Hyper-parameter	Value
Input	word, POS tag, Glove	100
	BERT	768
	dropout	0.33
CharCNN	kernel	[1,2,3,5]
	hidden size	25
	dropout	0.33
BiLSTM	#layer	6
	hidden size	400
	dropout	0.33
Xformer	#layer	6
	model size	200
	#head	8
	FeedForward size	800
	dropout	0.2
USE	#layer	6
	output size	256
	#head	8
	MLP size	800
	dropout	0.2
Trainer	optimizer	Adam
	learning rate	0.002
	(β_1, β_2)	(0.9, 0.9)

Table 7: Hyper-parameters for experiments.

E Structured-Dependent for All Steps

In Figure 7, we list the structured-dependent information for all steps on the example.