# Sequence-to-sequence Models for Cache Transition Systems

**Xiaochang Peng**[1], **Linfeng Song**[1], **Daniel Gildea**[1], **Giorgio Satta**[2]
[1]University of Rochester [2]University of Padua
{xpeng,lsong10,gildea}@cs.rochester.edu,
satta@dei.unipd.it

## Abstract

In this paper, we present a sequence-to-sequence based approach for mapping natural language sentences to AMR semantic graphs. We transform the sequence to graph mapping problem to a word sequence to transition action sequence problem using a special transition system called a cache transition system. To address the sparsity issue of neural AMR parsing, we feed feature embeddings from the transition state to provide relevant local information for each decoder state. We present a monotonic hard attention model for the transition framework to handle the strictly left-to-right alignment between each transition state and the current buffer input focus. We evaluate our neural transition model on the AMR parsing task, and our parser outperforms other sequence-to-sequence approaches and achieves competitive results in comparison with the best-performing models.[1]

## 1 Introduction

Abstract Meaning Representation (AMR) (Banarescu et al., 2013) is a semantic formalism where the meaning of a sentence is encoded as a rooted, directed graph. Figure 1 shows an example of an AMR in which the nodes represent the AMR concepts and the edges represent the relations between the concepts. AMR has been used in various applications such as text summarization (Liu et al., 2015), sentence compression (Takase et al., 2016), and event extraction (Huang et al., 2016).

[1]The implementation of our parser is available at https://github.com/xiaochang13/CacheTransition-Seq2seq

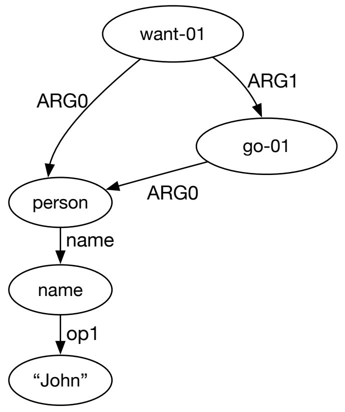

Figure 1: An example of AMR graph representing the meaning of: "John wants to go"

The task of AMR graph parsing is to map natural language strings to AMR semantic graphs. Different parsers have been developed to tackle this problem (Flanigan et al., 2014; Wang et al., 2015b,a; Peng et al., 2015; Artzi et al., 2015; Pust et al., 2015; van Noord and Bos, 2017). On the other hand, due to the limited amount of labeled data and the large output vocabulary, the sequence-to-sequence model has not been very successful on AMR parsing. Peng et al. (2017) propose a linearization approach that encodes labeled graphs as sequences. To address the data sparsity issue, low-frequency entities and tokens are mapped to special categories to reduce the vocabulary size for the neural models. Konstas et al. (2017) use self-training on a huge amount of unlabeled text to lower the out-of-vocabulary rate. However, the final performance still falls behind the best-performing models.

The best performing AMR parsers model graph structures directly. One approach to modeling graph structures is to use a transition system to build graphs step by step, as shown by the system of Wang and Xue (2017), which is currently the top performing system. This raises the question of whether the advantages of neural and transition-based system can be combined, as for example with the syntactic parser of Dyer et al. (2015), who use stack LSTMs to capture action history information in the transition state of the transition system. Ballesteros and Al-Onaizan (2017) apply stack-LSTM to transition-based AMR parsing and achieve competitive results, which shows that local transition state information is important for predicting transition actions.

Instead of linearizing the target AMR graph to a sequence structure, Buys and Blunsom (2017) propose a *sequence-to-action-sequence* approach where the reference AMR graph is replaced with an action derivation sequence by running a deterministic *oracle* algorithm on the training sentence, AMR graph pairs. They use a separate alignment probability to explicitly model the hard alignment from graph nodes to sentence tokens in the buffer.

Gildea et al. (2018) propose a special transition framework called a cache transition system to generate the set of semantic graphs. They adapt the stack-based parsing system by adding a working set, which they refer to as a cache, to the traditional stack and buffer. Peng et al. (2018) apply the cache transition system to AMR parsing and design refined action phases, each modeled with a separate feedforward neural network, to deal with some practical implementation issues.

In this paper, we propose a sequence-to-action-sequence approach for AMR parsing with cache transition systems. We want to take advantage of the sequence-to-sequence model to encode whole-sentence context information and the history action sequence, while using the transition system to constrain the possible output. The transition system can also provide better local context information than the linearized graph representation, which is important for neural AMR parsing given the limited amount of data.

More specifically, we use bi-LSTM to encode two levels of input information for AMR parsing: word level and concept level, each refined with more general category information such as lemmatization, POS tags, and concept categories.

We also want to make better use of the complex transition system to address the data sparsity issue for neural AMR parsing. We extend the hard attention model of Aharoni and Goldberg (2017), which deals with the nearly-monotonic alignment in the morphological inflection task, to the more general scenario of transition systems where the input buffer is processed from left-to-right. When we process the buffer in this ordered manner, the sequence of target transition actions are also strictly aligned left-to-right according to the input order. On the decoder side, we augment the prediction of output action with embedding features from the current transition state. Our experiments show that encoding information from the transition state significantly improves sequence-to-sequence models for AMR parsing.

## 2 Cache Transition Parser

We adopt the transition system of Gildea et al. (2018), which has been shown to have good coverage of the graphs found in AMR.

A **cache transition parser** consists of a stack, a cache, and an input buffer. The stack is a sequence $\sigma$ of (integer, concept) pairs, as explained below, with the topmost element always at the rightmost position. The buffer is a sequence of ordered concepts $\beta$ containing a suffix of the input concept sequence, with the first element to be read as a newly introduced concept/vertex of the graph. (We use the terms concept and vertex interchangeably in this paper.) Finally, the cache is a sequence of concepts $\eta = [v_1, \ldots, v_m]$. The element at the leftmost position is called the first element of the cache, and the element at the rightmost position is called the last element.

Operationally, the functioning of the parser can be described in terms of configurations and transitions. A **configuration** of our parser has the form:

$$C = (\sigma, \eta, \beta, G_p)$$

where $\sigma$, $\eta$ and $\beta$ are as described above, and $G_p$ is the partial graph that has been built so far. The initial configuration of the parser is $([], [\$, \ldots, \$], [c_1, \ldots, c_n], \emptyset)$, meaning that the stack and the partial graph are initially empty, and the cache is filled with $m$ occurrences of the special symbol $\$$. The buffer is initialized with all the graph vertices constrained by the order of the input sentence. The final configuration is $([], [\$, \ldots, \$], [], G)$, where the stack and the cache are as in the initial configuration and the buffer is empty. The constructed graph is the target AMR graph.

| stack | cache | buffer | edges | actions taken |
|---|---|---|---|---|
| [] | [\$, \$, \$] | [Per, want-01, go-01] | $\emptyset$ | — |
| [1, \$] | [\$, \$, Per] | [want-01, go-01] | $\emptyset$ | *Shift*; *PushIndex*(1) |
| [1, \$] | [\$, \$, Per] | [want-01, go-01] | $\emptyset$ | *Arc(1, -, NULL)*; *Arc(2, -, NULL)* |
| [1, \$, 1, \$] | [\$, Per, want-01] | [go-01] | $\emptyset$ | *Shift*; *PushIndex*(1) |
| [1, \$, 1, \$] | [\$, Per, want-01] | [go-01] | $E_1$ | *Arc(1, -, NULL)*; *Arc(2, L, ARG0)* |
| [1, \$, 1, \$, 1, \$] | [Per, want-01, go-01] | [] | $E_1$ | *Shift*; *PushIndex*(1) |
| [1, \$, 1, \$, 1, \$] | [Per, want-01, go-01] | [] | $E_2$ | *Arc(1, L, ARG0)*; *Arc(2, R, ARG1)* |
| [1, \$, 1, \$] | [\$, Per, want-01 ] | [] | $E_2$ | *Pop* |
| [1, \$] | [\$, \$, Per] | [] | $E_2$ | *Pop* |
| [] | [\$, \$, \$] | [] | $E_2$ | *Pop* |

Figure 2: Example run of the cache transition system constructing the graph for the sentence "John wants to go" with cache size of 3. The left four columns show the parser configurations after taking the actions shown in the last column. $E_1 = \{(Per, want\text{-}01, L\text{-}ARG0)\}$, $E_2 = \{(Per, want\text{-}01, L\text{-}ARG0), (Per, go\text{-}01, L\text{-}ARG0), (want\text{-}01, go\text{-}01, R\text{-}ARG1)\}$.

In the first step, which is called concept identification, we map the input sentence $w_{1:n'} = w_1, \ldots, w_{n'}$ to a sequence of concepts $c_{1:n} = c_1, \ldots, c_n$. We decouple the problem of concept identification from the transition system and initialize the buffer with a recognized concept sequence from another classifier, which we will introduce later. As the sequence-to-sequence model uses all possible output actions as the target vocabulary, this can significantly reduce the target vocabulary size. The **transitions** of the parser are specified as follows.

1. *Pop* pops a pair $(i, v)$ from the stack, where the integer $i$ records the position in the cache that it originally came from. We place concept $v$ in position $i$ in the cache, shifting the remainder of the cache one position to the right, and discarding the last element in the cache.
2. *Shift* signals that we will start processing the next input concept, which will become a new vertex in the output graph.
3. *PushIndex*$(i)$ shifts the next input concept out of the buffer and moves it into the last position of the cache. We also take out the concept $v_i$ appearing at position $i$ in the cache and push it onto the stack $\sigma$, along with the integer $i$ recording its original position in the cache.[2]
4. *Arc(i, d, l)* builds an arc with direction $d$ and label $l$ between the rightmost concept and the $i$-th concept in the cache. The label $l$ is *NULL* if no arc is made and we use the action *NOARC* in this case. Otherwise we decompose the arc decision into two actions *ARC* and *d-l*. We consider all arc decisions between the rightmost cache concept and each of the other concepts in the cache. We can consider this phase as first making a binary decision whether there is an arc, and then predicting the label in case there is one, between each concept pair.

[2]Our transition design is different from Peng et al. (2018) in two ways: the PushIndex phase is initiated before making all the arc decisions; the newly introduced concept is placed at the last cache position instead of the leftmost buffer position, which essentially increases the cache size by 1.

Given the sentence "John wants to go" and the recognized concept sequence "Per want-01 go-01" (person name category *Per* for "John"), our cache transition parser can construct the AMR graph shown in Figure 1 using the run shown in Figure 2 with cache size of 3.

## 2.1 Oracle Extraction Algorithm

We use the following **oracle** algorithm (Nivre, 2008) to derive the sequence of actions that leads to the gold AMR graph for a cache transition parser with cache size $m$. The correctness of the oracle is shown by Gildea et al. (2018).

Let $E_G$ be the set of edges of the gold graph $G$. We maintain the set of vertices that is not yet shifted into the cache as $S$, which is initialized with all vertices in $G$. The vertices are ordered according to their aligned position in the word sequence and the unaligned vertices are listed according to their order in the depth-first traversal of the graph. The oracle algorithm can look into

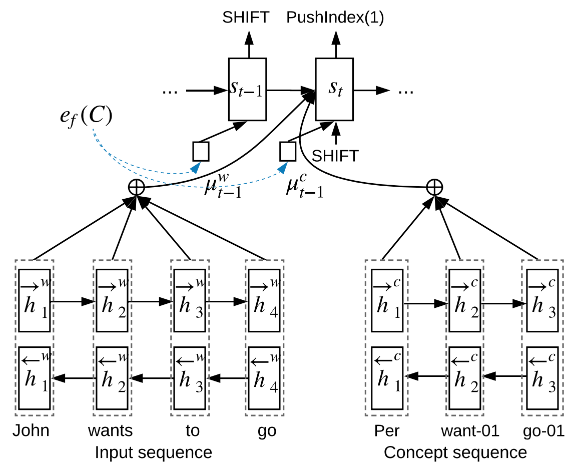

Figure 3: Sequence-to-sequence model with soft attention, encoding a word sequence and concept sequence separately by two BiLSTM encoders.

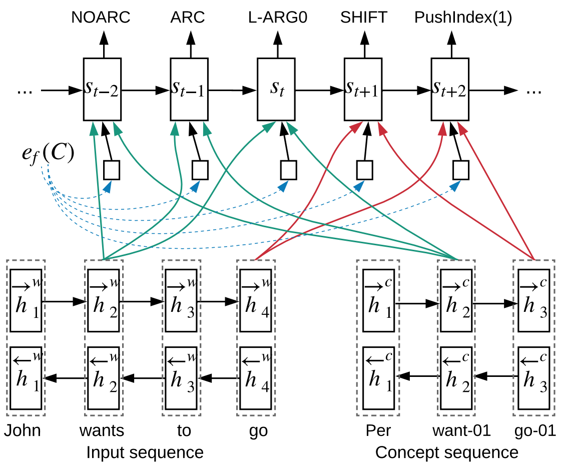

Figure 4: Sequence-to-sequence model with monotonic hard attention. Different colors show the changes of hard attention focus.

$E_G$ to decide which transition to take next, or else to decide that it should fail. This decision is based on the mutually exclusive rules listed below.

1. ShiftOrPop phase: the oracle chooses transition *Pop*, in case there is no edge $(v_m, v)$ in $E_G$ such that vertex $v$ is in $S$, or chooses transition *Shift* and proceeds to the next phase.
2. PushIndex phase: in this phase, the oracle first chooses a position $i$ (as explained below) in the cache to place the candidate concept and removes the vertex at this position and places its index, vertex pair onto the stack. The oracle chooses transition *PushIndex(i)* and proceeds to the next phase.
3. ArcBinary, ArcLabel phases: between the rightmost cache concept and each concept in the cache, we make a binary decision about whether there is an arc between them. If there is an arc, the oracle chooses its direction and label. After arc decisions to $m-1$ cache concepts are made, we jump to the next step.
4. If the stack and buffer are both empty, and the cache is in the initial state, the oracle finishes with success, otherwise we proceed to the first step.

We use the equation below to choose the cache concept to take out in the step *PushIndex(i)*. For $j \in [|\beta|]$, we write $\beta_j$ to denote the $j$-th vertex in $\beta$. We choose a vertex $v_{i^*}$ in $\eta$ such that:

$$i^* = \underset{i \in [m]}{\operatorname{argmax}} \min \{j \mid (v_i, \beta_j) \in E_G\}$$

In words, $v_{i^*}$ is the concept from the cache whose closest neighbor in the buffer $\beta$ is furthest forward in $\beta$. We move out of the cache vertex $v_{i^*}$ and push it onto the stack, for later processing.

For each training example $(x_{1:n}, g)$, the transition system generates the output AMR graph $g$ from the input sequence $x_{1:n}$ through an oracle sequence $a_{1:q} \in \Sigma_a^*$, where $\Sigma_a$ is the union of all possible actions. We model the probability of the output with the action sequence:

$$P(a_{1:q}|x_{1:n}) = \prod_{t=1}^{q} P(a_t|a_1, \ldots, a_{t-1}, x_{1:n}; \theta)$$

which we estimate using a sequence-to-sequence model, as we will describe in the next section.

## 3 Soft vs Hard Attention for Sequence-to-action-sequence

Shown in Figure 3, our sequence-to-sequence model takes a word sequence $w_{1:n'}$ and its mapped concept sequence $c_{1:n}$ as the input, and the action sequence $a_{1:q}$ as the output. It uses two BiLSTM encoders, each encoding an input sequence. As the two encoders have the same structure, we only introduce the encoder for the word sequence in detail below.

### 3.1 BiLSTM Encoder

Given an input word sequence $w_{1:n'}$, we use a bidirectional LSTM to encode it. At each step $j$, the current hidden states $\overleftarrow{h}^w_j$ and $\overrightarrow{h}^w_j$ are generated from the previous hidden states $\overleftarrow{h}^w_{j+1}$ and $\overrightarrow{h}^w_{j-1}$, and the representation vector $x_j$ of the current input word $w_j$:

$$\overleftarrow{h}^w_j = \text{LSTM}(\overleftarrow{h}^w_{j+1}, x_j)$$
$$\overrightarrow{h}^w_j = \text{LSTM}(\overrightarrow{h}^w_{j-1}, x_j)$$

The representation vector $x_j$ is the concatenation of the embeddings of its word, lemma, and POS tag, respectively. Then the hidden states of both directions are concatenated as the final hidden state for word $w_j$:

$$h^w_j = [\overleftarrow{h}^w_j; \overrightarrow{h}^w_j]$$

Similarly, for the concept sequence, the final hidden state for concept $c_j$ is:

$$h^c_j = [\overleftarrow{h}^c_j; \overrightarrow{h}^c_j]$$

## 3.2 LSTM Decoder with Soft Attention

We use an attention-based LSTM decoder (Bahdanau et al., 2014) with two attention memories $H_w$ and $H_c$, where $H_w$ is the concatenation of the state vectors of all input words, and $H_c$ for input concepts correspondingly:

$$H_w = [h^w_1; h^w_2; \ldots; h^w_{n'}] \quad (1)$$
$$H_c = [h^c_1; h^c_2; \ldots; h^c_n] \quad (2)$$

The decoder yields an action sequence $a_1, a_2, \ldots, a_q$ as the output by calculating a sequence of hidden states $s_1, s_2 \ldots, s_q$ recurrently. While generating the $t$-th output action, the decoder considers three factors: (1) the previous hidden state of the LSTM model $s_{t-1}$; (2) the embedding of the previous generated action $e_{t-1}$; and (3) the previous context vectors for words $\mu^w_{t-1}$ and concepts $\mu^c_{t-1}$, which are calculated using $H_w$ and $H_c$, respectively. When $t = 1$, we initialize $\mu_0$ as a zero vector, and set $e_0$ to the embedding of the start token "$\langle s \rangle$". The hidden state $s_0$ is initialized as:

$$s_0 = W_d[\overleftarrow{h}^w_1; \overrightarrow{h}^w_n; \overleftarrow{h}^c_1; \overrightarrow{h}^c_n] + b_d,$$

where $W_d$ and $b_d$ are model parameters.

For each time-step $t$, the decoder feeds the concatenation of the embedding of previous action $e_{t-1}$ and the previous context vectors for words $\mu^w_{t-1}$ and concepts $\mu^c_{t-1}$ into the LSTM model to update its hidden state.

$$s_t = \text{LSTM}(s_{t-1}, [e_{t-1}; \mu^w_{t-1}; \mu^c_{t-1}]) \quad (3)$$

Then the attention probabilities for the word sequence and the concept sequence are calculated similarly. Take the word sequence as an example, $\alpha^w_{t,i}$ on $h^w_i \in H_w$ for time-step $t$ is calculated as:

$$\epsilon_{t,i} = v_c^T \tanh(W_h h^w_i + W_s s_t + b_c)$$
$$\alpha^w_{t,i} = \frac{\exp(\epsilon_{t,i})}{\sum_{j=1}^{N} \exp(\epsilon_{t,j})}$$

$W_h$, $W_s$, $v_c$ and $b_c$ are model parameters. The new context vector $\mu^w_t = \sum_{i=1}^{n} \alpha^w_{t,i} h^w_i$. The calculation of $\mu^c_t$ follows the same procedure, but with a different set of model parameters.

The output probability distribution over all actions at the current state is calculated by:

$$P_{\Sigma_a} = \text{softmax}(V_a[s_t; \mu^w_t; \mu^c_t] + b_a), \quad (4)$$

where $V_a$ and $b_a$ are learnable parameters, and the number of rows in $V_a$ represents the number of all actions. The symbol $\Sigma_a$ is the set of all actions.

## 3.3 Monotonic Hard Attention for Transition Systems

When we process each buffer input, the next few transition actions are closely related to this input position. The buffer maintains the order information of the input sequence and is processed strictly left-to-right, which essentially encodes a monotone alignment between the transition action sequence and the input sequence.

As we have generated a concept sequence from the input word sequence, we maintain two hard attention pointers, $l_w$ and $l_c$, to model monotonic attention to word and concept sequences respectively. The update to the decoder state now relies on a single position of each input sequence in contrast to Equation 3:

$$s_t = \text{LSTM}(s_{t-1}, [e_{t-1}; h^w_{l_w}; h^c_{l_c}]) \quad (5)$$

**Control Mechanism.** Both pointers are initialized as 0 and advanced to the next position deterministically. We move the concept attention focus $l_c$ to the next position after arc decisions to all the other $m - 1$ cache concepts are made. We move the word attention focus $l_w$ to its aligned position in case the new concept is aligned, otherwise we don't move the word focus. As shown in Figure 4, after we have made arc decisions from concept *want-01* to the other cache concepts, we move the concept focus to the next concept *go-01*. As this concept is aligned, we move the word focus to its aligned position *go* in the word sequence and skip the unaligned word *to*.

### 3.4 Transition State Features for Decoder

Another difference of our model with Buys and Blunsom (2017) is that we extract features from the current transition state configuration $C_t$:

$$e_f(C_t) = [e_{f_1}(C_t); e_{f_2}(C_t); \cdots; e_{f_l}(C_t)]$$

where $l$ is the number of features extracted from $C_t$ and $e_{f_k}(C_t)$ $(k = 1, \ldots, l)$ represents the embedding for the $k$-th feature, which is learned during training. These feature embeddings are concatenated as $e_f(C_t)$, and fed as additional input to the decoder. For the soft attention decoder:

$$s_t = \text{LSTM}(s_{t-1}, [e_{t-1}; \mu^w_{t-1}; \mu^c_{t-1}; e_f(C_t)])$$

and for the hard attention decoder:

$$s_t = \text{LSTM}(s_{t-1}, [e_{t-1}; h^w_{l_w}; h^c_{l_c}; e_f(C_t)])$$

We use the following features in our experiments:

1. Phase type: indicator features showing which phase the next transition is.
2. ShiftOrPop features: *token features*[3] for the rightmost cache concept and the leftmost buffer concept. Number of dependencies to words on the right, and the top three dependency labels for them.
3. ArcBinary or ArcLabel features: token features for the rightmost concept and the current cache concept it makes arc decisions to. Word, concept and dependency distance between the two concepts. The labels for the two most recent outgoing arcs for these two concepts and their first incoming arc and the number of incoming arcs. Dependency label between the two positions if there is a dependency arc between them.
4. PushIndex features: token features for the leftmost buffer concept and all the concepts in the cache.

The phase type features are deterministic from the last action output. For example, if the last action output is *Shift*, the current phase type would be *PushIndex*. We only extract corresponding features for this phase and fill all the other feature types with *-NULL-* as placeholders. The features for other phases are similar.

[3]Concept, concept category at the specified position in concept sequence. And the word, lemma, POS tag at the aligned input position.

## 4 AMR Parsing

### 4.1 Training and Decoding

We train our models using the cross-entropy loss, over each oracle action sequence $a^*_1, \ldots, a^*_q$:

$$L = -\sum_{t=1}^{q} \log P(a^*_t | a^*_1, \ldots, a^*_{t-1}, X; \theta), \quad (6)$$

where $X$ represents the input word and concept sequences, and $\theta$ is the model parameters. Adam (Kingma and Ba, 2014) with a learning rate of 0.001 is used as the optimizer, and the model that yields the best performance on the dev set is selected to evaluate on the test set. Dropout with rate 0.3 is used during training. Beam search with a beam size of 10 is used for decoding. Both training and decoding use a Tesla K20X GPU.

Hidden state sizes for both encoder and decoder are set to 100. The word embeddings are initialized from Glove pretrained word embeddings (Pennington et al., 2014) on Common Crawl, and are not updated during training. The embeddings for POS tags and features are randomly initialized, with the sizes of 20 and 50, respectively.

### 4.2 Preprocessing and Postprocessing

As the AMR data is very sparse, we collapse some subgraphs or spans into categories based on the alignment. We define some special categories such as named entities (*NE*), dates (*DATE*), single rooted subgraphs involving multiple concepts (*MULT*)[4], numbers (*NUMBER*) and phrases (*PHRASE*). The phrases are extracted based on the multiple-to-one alignment in the training data. One example phrase is *more than* which aligns to a single concept *more-than*. We first collapse spans and subgraphs into these categories based on the alignment from the JAMR aligner (Flanigan et al., 2014), which greedily aligns a span of words to AMR subgraphs using a set of heuristics. This categorization procedure enables the parser to capture mappings from continuous spans on the sentence side to connected subgraphs on the AMR side.

We use the semi-Markov model from Flanigan et al. (2016) as the concept identifier, which jointly segments the sentence into a sequence of spans and maps each span to a subgraph. During decoding, our output has categories, and we need to map

[4]For example, verbalization of “teacher” as “(person :ARG0-of teach-01)”, or “minister” as “(person :ARG0-of (have-org-role-91 :ARG2 minister))”.

| | ShiftOrPop | PushIndex | ArcBinary | ArcLabel |
|---|---|---|---|---|
| Peng et al. (2018) | 0.87 | **0.87** | 0.83 | **0.81** |
| Soft+feats | 0.93 | 0.84 | 0.91 | 0.75 |
| Hard+feats | **0.94** | 0.85 | **0.93** | 0.77 |

Table 1: Performance breakdown of each transition phase.

each category to the corresponding AMR concept or subgraph. We save a table $Q$ which shows the original subgraph each category is collapsed from, and map each category to its original subgraph representation. We also use heuristic rules to generate the target-side AMR subgraph representation for *NE*, *DATE*, and *NUMBER* based on the source side tokens.

## 5 Experiments

We evaluate our system on the released dataset (LDC2015E86) for SemEval 2016 task 8 on meaning representation parsing (May, 2016). The dataset contains 16,833 training, 1,368 development, and 1,371 test sentences which mainly cover domains like newswire, discussion forum, etc. All parsing results are measured by Smatch (version 2.0.2) (Cai and Knight, 2013).

### 5.1 Experiment Settings

We categorize the training data using the automatic alignment and dump a template for date entities and frequent phrases from the multiple to one alignment. We also generate an alignment table from tokens or phrases to their candidate target-side subgraphs. For the dev and test data, we first extract the named entities using the Illinois Named Entity Tagger (Ratinov and Roth, 2009) and extract date entities by matching spans with the date template. We further categorize the dataset with the categories we have defined. After categorization, we use Stanford CoreNLP (Manning et al., 2014) to get the POS tags and dependencies of the categorized dataset. We run the oracle algorithm separately for training and dev data (with alignment) to get the statistics of individual phases. We use a cache size of 5 in our experiments.

### 5.2 Results

**Individual Phase Accuracy** We first evaluate the prediction accuracy of individual phases on the dev oracle data assuming gold prediction history. The four transition phases *ShiftOrPop*, *PushIndex*, *ArcBinary*, and *ArcLabel* account for 25%, 12.5%, 50.1%, and 12.4% of the total transition actions respectively. Table 1 shows the phase-wise accuracy of our sequence-to-sequence model. Peng et al. (2018) use a separate feedforward network to predict each phase independently. We use the same alignment from the SemEval dataset as in Peng et al. (2018) to avoid differences resulting from the aligner. *Soft+feats* shows the performance of our sequence-to-sequence model with soft attention and transition state features, while *Hard+feats* is using hard attention. We can see that the hard attention model outperforms the soft attention model in all phases, which shows that the single-pointer attention finds more relevant information than the soft attention on the relatively small dataset. The sequence-to-sequence models perform better than the feedforward model of Peng et al. (2018) on *ShiftOrPop* and *ArcBinary*, which shows that the whole-sentence context information is important for the prediction of these two phases. On the other hand, the sequence-to-sequence models perform worse than the feedforward models on *PushIndex* and *ArcLabel*. One possible reason is that the model tries to optimize the overall accuracy, while these two phases account for fewer than 25% of the total transition actions and might be less attended to during the update.

**Impact of Different Components** Table 2 shows the impact of different components for the sequence-to-sequence model. We can see that the transition state features play a very important role for predicting the correct transition action. This is because different transition phases have very different prediction behaviors and need different types of local information for the prediction. Relying on the sequence-to-sequence model alone does not perform well in disambiguating these choices, while the transition state can enforce direct constraints. We can also see that while the hard attention only attends to one position of the input, it performs slightly better than the soft attention model, while the time complexity is lower.

**Impact of Different Cache Sizes** The cache size of the transition system can be optimized as a trade-off between coverage of AMR graphs and the prediction accuracy. While larger cache size increases the coverage of AMR graphs, it complicates the prediction procedure with more cache decisions to make. From Table 3 we can see that

| System | P | R | F |
|---|---|---|---|
| Soft | 0.55 | 0.51 | 0.53 |
| Soft+feats | 0.69 | 0.63 | 0.66 |
| Hard+feats | 0.70 | 0.64 | 0.67 |

Table 2: Impact of various components for the sequence-to-sequence model (dev).

| Cache Size | P | R | F |
|---|---|---|---|
| 4 | 0.69 | 0.63 | 0.66 |
| 5 | 0.70 | 0.64 | 0.67 |
| 6 | 0.69 | 0.64 | 0.66 |

Table 3: Impact of cache size for the sequence-to-sequence model, hard attention (dev).

the hard attention model performs best with cache size 5. The soft attention model also achieves best performance with the same cache size.

**Comparison with other Parsers** Table 4 shows the comparison with other AMR parsers. The first three systems are some competitive neural models. We can see that our parser significantly outperforms the sequence-to-action-sequence model of Buys and Blunsom (2017). Konstas et al. (2017) use a linearization approach that linearizes the AMR graph to a sequence structure and use self-training on 20M unlabeled Gigaword sentences. Our model achieves better results without using additional unlabeled data, which shows that relevant information from the transition system is very useful for the prediction. Our model also outperforms the stack-LSTM model by Ballesteros and Al-Onaizan (2017), while their model is evaluated on the previous release of LDC2014T12.

| System | P | R | F |
|---|---|---|---|
| Buys and Blunsom (2017) | – | – | 0.60 |
| Konstas et al. (2017) | 0.60 | 0.65 | 0.62 |
| Ballesteros and Al-Onaizan (2017)* | – | – | 0.64 |
| Damonte et al. (2017) | – | – | 0.64 |
| Peng et al. (2018) | 0.69 | 0.59 | 0.64 |
| Wang et al. (2015b) | 0.64 | 0.62 | 0.63 |
| Wang et al. (2015a) | 0.70 | 0.63 | 0.66 |
| Flanigan et al. (2016) | 0.70 | 0.65 | 0.67 |
| Wang and Xue (2017) | 0.72 | 0.65 | 0.68 |
| Ours soft attention | 0.68 | 0.63 | 0.65 |
| Ours hard attention | 0.69 | 0.64 | 0.66 |

Table 4: Comparison to other AMR parsers. *Model has been trained on the previous release of the corpus (LDC2014T12).

| System | P | R | F |
|---|---|---|---|
| Peng et al. (2018) | 0.44 | 0.28 | 0.34 |
| Damonte et al. (2017) | – | – | 0.41 |
| JAMR | 0.47 | 0.38 | 0.42 |
| Ours | 0.58 | 0.34 | 0.43 |

Table 5: Reentrancy statistics.

We also show the performance of some of the best-performing models. While our hard attention achieves slightly lower performance in comparison with Wang et al. (2015a) and Wang and Xue (2017), it is worth noting that their approaches of using WordNet, semantic role labels and word cluster features are complimentary to ours. The alignment from the aligner and the concept identification identifier also play an important role for improving the performance. Wang and Xue (2017) propose to improve AMR parsing by improving the alignment and concept identification, which can also be combined with our system to improve the performance of a sequence-to-sequence model.

**Dealing with Reentrancy** Reentrancy is an important characteristic of AMR, and we evaluate the Smatch score only on the reentrant edges following Damonte et al. (2017). From Table 5 we can see that our hard attention model significantly outperforms the feedforward model of Peng et al. (2018) in predicting reentrancies. This is because predicting reentrancy is directly related to the *ArcBinary* phase of the cache transition system since it decides to make multiple arc decisions to the same vertex, and we can see from Table 1 that the hard attention model has significantly better prediction accuracy in this phase. We also compare the reentrancy results of our transition system with two other systems, Damonte et al. (2017) and JAMR, where these statistics are available. From Table 5, we can see that our cache transition system slightly outperforms these two systems in predicting reentrancies.

Figure 5 shows a reentrancy example where JAMR and the feedforward network of Peng et al. (2018) do not predict well, while our system predicts the correct output. JAMR fails to predict the reentrancy arc from *desire-01* to *i*, and connects the wrong arc from “live-01” to “-” instead of from “desire-01”. The feedforward model of Peng et al. (2018) fails to predict the two arcs from *desire-01*

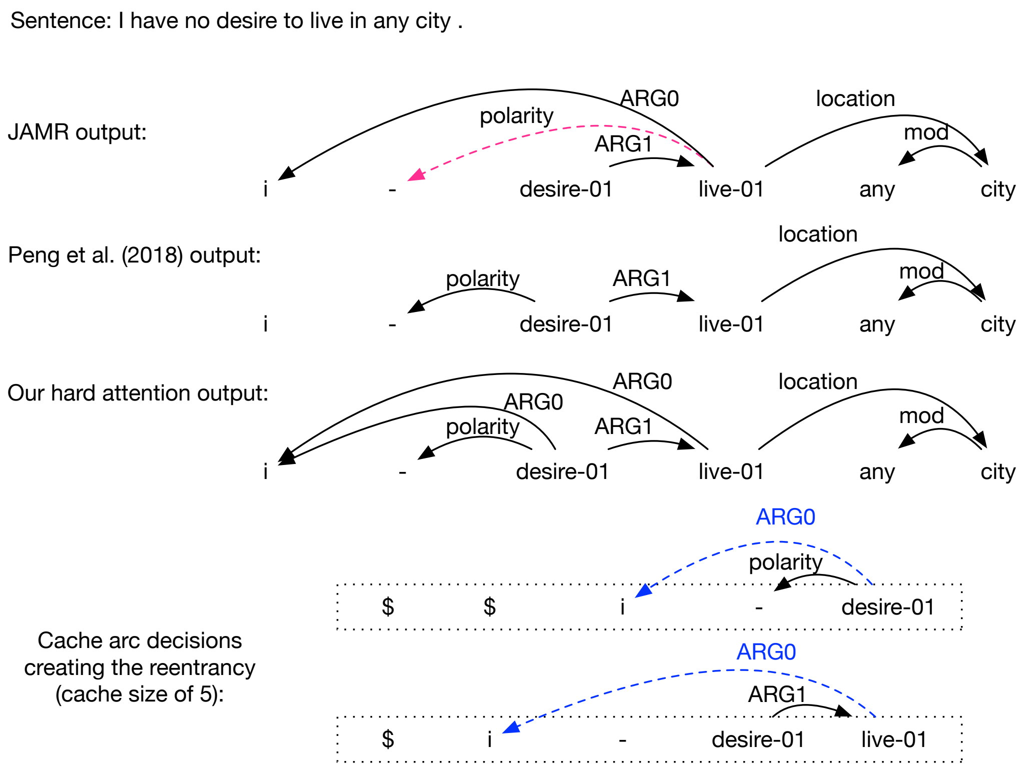
Sentence: I have no desire to live in any city .
JAMR output:
Peng et al. (2018) output:
Our hard attention output:
Cache arc decisions creating the reentrancy (cache size of 5):
i
-
desire-01
live-01
any
city
ARG0
ARG1
polarity
location
mod
$

Figure 5: An example showing how our system predicts the correct reentrancy.

and *live-01* to *i*. This error is because their feed-forward ArcBinary classifier does not model long-term dependency and usually prefers making arcs between words that are close and not if they are distant. Our classifier, which encodes both word and concept sequence information, can accurately predict the reentrancy through the two arc decisions shown in Figure 5. When *desire-01* and *live-01* are shifted into the cache respectively, the transition system makes a left-going arc from each of them to the same concept *i*, thus creating the reentrancy as desired.

## 6 Conclusion

In this paper, we have presented a sequence-to-action-sequence approach for cache transition systems and applied it to AMR parsing. To address the data sparsity issue for neural AMR parsing, we show that the transition state features are very helpful in constraining the possible output and improving the performance of sequence-to-sequence models. We also show that the monotonic hard attention model can be generalized to the transition-based framework and outperforms the soft attention model when limited data is available. While we are focused on AMR parsing in this paper, in future work our cache transition system and the presented sequence-to-sequence models can be potentially applied to other semantic graph parsing tasks (Oepen et al., 2015; Du et al., 2015; Zhang et al., 2016; Cao et al., 2017).

## Acknowledgments

We gratefully acknowledge the assistance of Hao Zhang from Google, New York for the monotonic hard attention idea and the helpful comments and suggestions.

## References

Roee Aharoni and Yoav Goldberg. 2017. Morphological inflection generation with hard monotonic attention. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2004–2015, Vancouver, Canada. Association for Computational Linguistics.

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710, Lisbon, Portugal. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Miguel Ballesteros and Yaser Al-Onaizan. 2017. AMR parsing using stack-LSTMs. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1269–1275.

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*, pages 178–186, Sofia, Bulgaria.

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 (Volume 1: Long Papers)*, volume 1, pages 1215–1226.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, pages 748–752.

Junjie Cao, Sheng Huang, Weiwei Sun, and Xiaojun Wan. 2017. Parsing to 1-endpoint-crossing, pagenumber-2 graphs. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2110–2120.

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, Long Papers*, volume 1, pages 536–546.

Yantao Du, Fan Zhang, Xun Zhang, Weiwei Sun, and Xiaojun Wan. 2015. Peking: Building semantic dependency graphs with a hybrid parser. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 927–931.

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 (Volume 1: Long Papers)*, volume 1, pages 334–343.

Jeffrey Flanigan, Chris Dyer, Noah A Smith, and Jaime Carbonell. 2016. CMU at SemEval-2016 task 8: Graph-based AMR parsing with infinite ramp loss. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1202–1206.

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1426–1436, Baltimore, Maryland.

Daniel Gildea, Giorgio Satta, and Xiaochang Peng. 2018. Cache transition systems for graph parsing. *Computational Linguistics*, 44(1):85–118.

Lifu Huang, Taylor Cassidy, Xiaocheng Feng, Heng Ji, Clare R Voss, Jiawei Han, and Avirup Sil. 2016. Liberal event extraction and event schema induction. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 258–268.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. 2017. Neural AMR: Sequence-to-sequence models for parsing and generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 146–157, Vancouver, Canada. Association for Computational Linguistics.

Fei Liu, Jeffrey Flanigan, Sam Thomson, Norman Sadeh, and Noah A Smith. 2015. Toward abstractive summarization using semantic representations. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1077–1086.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60.

Jonathan May. 2016. SemEval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1063–1073, San Diego, California.

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

Rik van Noord and Johan Bos. 2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *arXiv preprint arXiv:1705.09980*.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinkova, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 915–926, Denver, Colorado.

Xiaochang Peng, Daniel Gildea, and Giorgio Satta. 2018. AMR parsing with cache transition systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-18)*.

Xiaochang Peng, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning (CoNLL-15)*, pages 32–41, Beijing, China.

Xiaochang Peng, Chuan Wang, Daniel Gildea, and Nianwen Xue. 2017. Addressing the data sparsity issue in neural AMR parsing. In *Proceedings of the European Chapter of the ACL (EACL-17)*.

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

Michael Pust, Ulf Hermjakob, Kevin Knight, Daniel Marcu, and Jonathan May. 2015. Parsing English into abstract meaning representation using syntax-based machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1143–1154, Lisbon, Portugal. Association for Computational Linguistics.

Lev Ratinov and Dan Roth. 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009)*, pages 147–155, Boulder, Colorado. Association for Computational Linguistics.

Sho Takase, Jun Suzuki, Naoaki Okazaki, Tsutomu Hirao, and Masaaki Nagata. 2016. Neural headline generation on abstract meaning representation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1054–1059.

Chuan Wang and Nianwen Xue. 2017. Getting the most out of AMR parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1257–1268.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015a. Boosting transition-based AMR parsing with refined actions and auxiliary analyzers. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 857–862, Beijing, China.

Chuan Wang, Nianwen Xue, and Sameer Pradhan. 2015b. A transition-based algorithm for AMR parsing. In *Proceedings of the 2015 Meeting of the North American chapter of the Association for Computational Linguistics (NAACL-15)*, pages 366–375, Denver, Colorado.

Xun Zhang, Yantao Du, Weiwei Sun, and Xiaojun Wan. 2016. Transition-based parsing for deep dependency structures. *Computational Linguistics*, 42(3):353–389.