Dealing with Co-reference in Neural Semantic Parsing

Rik van Noord
CLCG, University of Groningen
r.i.k.van.noord@rug.nl

Johan Bos
CLCG, University of Groningen
johan.bos@rug.nl

Abstract

Linguistic phenomena like pronouns, control constructions, or co-reference give rise to co-indexed variables in meaning representations. We review three different methods for dealing with co-indexed variables in the output of neural semantic parsing of abstract meaning representations: (a) copying concepts during training and restoring co-indexation in a post-processing step; (b) explicit indexing of co-indexation; and (c) using absolute paths to designate co-indexing. The second method gives the best results and outperforms the baseline by 2.9 F-score points.

1 Introduction

Semantic parsing is the task of mapping a natural language sentence into a meaning representation (a logical form). One of the problems a semantic parser has to deal with is co-indexed variables, which arise in antecedent-anaphor relations, proper name co-reference, control constructions and other linguistic phenomena. Examples of such constructions are given in (1)–(4):

(1) Bob$_i$ likes himself$_i$.
(2) Jack$_i$ wants ϵ_i to buy an ice-cream.
(3) Peter$_i$ sold his$_i$ car.
(4) Sue$_i$ saw Mary$_j$, but Mary$_j$ did not see Sue$_i$.

In the context of this paper, we represent meanings using the formalism of Abstract Meaning Representation (AMR), as introduced by Banarescu et al. (2013). AMRs can be seen as graphs connecting concepts by relations. Each concept is represented by a named instance. Co-reference is established by re-using these instances. For example, the AMRs corresponding to examples (1) and (2) above are given in Figure 1. Note that, due to the bracketing, the variable *b* encapsulates the whole entity `person :name "Bob"` and not just `person`, i.e. *b* stands for a person with the name Bob.

```
(l / like-01
   :ARG0 (b / person :name "Bob")
   :ARG1 b)
```

```
(w / want-01
   :ARG1 (p / person :name "Jack")
   :ARG3 (b / buy-01
            :ARG1 p
            :ARG2 (i / ice-cream)))
```

Figure 1: AMRs representing the meaning of examples (1) and (2).

That there is a lot to gain in this area can be seen by applying the AMR evaluation suite of Damonte et al. (2017), which calculates nine different metrics to evaluate AMR parsing, *reentrancy* being one of them. Out of the four systems that made these scores available (all scores reported in van Noord and Bos (2017)), the reentrancy metric obtained the lowest F-score for three of them.

Various methods have been proposed to automatically parse AMRs, ranging from syntax-based approaches (e.g. Flanigan et al. (2014); Wang et al. (2015); Pust et al. (2015); Damonte et al. (2017)) to the more recent neural approaches (Peng et al. (2017); Buys and Blunsom (2017); Konstas et al. (2017); Foland and Martin (2017); van Noord and Bos (2017)). Especially the neural approaches are interesting, since they all use some sort of linearization method and therefore need a predefined way to handle reentrancy. Peng et al. (2017) and Buys and Blunsom (2017) use a special character to indicate reentrancy and restore co-referring variables in a post-processing step. Konstas et al. (2017) simply replace reentrancy variables by their co-referring concept in the input and never outputs co-referring nodes. Foland and Martin (2017) and van Noord and Bos (2017) use the same input transformation as Konstas et al. (2017), but do try to restore co-referring nodes by merging all equal concepts into a single concept in a post-processing step. All these methods have in common that they are not very sophisticated, but more importantly, that it is not clear what the exact impact of these methods is on the final performance of the model, making it unclear what the best implementation is for future neural AMR parsers.

In this paper we present three methods to handle reentrancy for AMR parsing. The first two methods are based on the previous work described above, while the third is a new, more principled method. These methods are applied on the model that reported the best results in the literature, the character-level neural semantic parsing method of van Noord and Bos (2017). In a nutshell, this method uses a character-based sequence-to-sequence model to translate sentences to AMRs. To enable this process, pre-processing and post-processing steps are needed. The aim of this paper is to find the best method to handle reentrancy in neural semantic parsing and to show the specific impact that each of the methods have on general performance.

2 Abstract Meaning Representations

AMRs, as introduced by Banarescu et al. (2013), are acyclic, directed graphs that show a representation of the meaning of a sentence. There are three ways to display an AMR: as a graph, as a tree, or as a set of triples. The AMRs in Figure 1 are shown as trees, which is also the input for our neural semantic parser. AMR concepts (e.g. like, person) are relating to each other by the use of two-place predicates (e.g. `:ARG0, :ARG1, :name`). For example, in the right AMR in Figure 1, *want* and *buy* are connected by the `:ARG3` predicate.

For our experiments, we use the annotated AMR corpus LDC2016E25, which contains 36,521 training, 1,368 development and 1,371 test AMRs. Co-indexed variables occur frequently in this data set, as can be seen in Table 1. About half of the AMRs contain at least one co-indexed variable, while about 20% of the total number of triples[1] contains a variable that has at least one anaphor in the AMR. This is the number of triples that is used in the reentrancy evaluation metric by Damonte et al. (2017). This number, however, also includes triples that would be present regardless of whether the variable is co-indexed (e.g. the instance triple of the antecedent). A more fair number might be the relation triples that contain an anaphor variable, which is exactly equal to the number of total co-indexed variables. This is still 3.4 to 4.2% of all triples in the data set.

Table 1: Statistics about the co-indexed variables in the AMR data set.

	Num AMRs	**Num AMRs with reentrancy**	**Total number of reentrancies**	**Total triples**	**Triples containing co-indexed var**
Training	36,520	17,589	40,582	968,512	189,426
Dev	1,368	706	1,590	46,737	8,704
Test	1,371	749	2,033	48,252	9,686
Silver	100,00	16,235	18,865	3,001,169	109,676

[1]Triples are used to evaluate AMR parsing systems, and are explained in Cai and Knight (2013).

3 Method

3.1 Variable-free AMRs

The actual values of the variables for AMR instances are unimportant. Hence, one can rename variables in an AMR as long as re-occurrences of variables are preserved. If variables are used only once in an AMR, they can therefore be completely eliminated from an AMR. This insight was used by Barzdins and Gosko (2016) in the first approach to neural semantic parsing of AMRs, because particular names of variables are very hard to learn for a neural model given the limited amount of data available. We present three ways to encode co-reference in variable-free AMRs.[2]

Method 1A: Baseline Note that if there are co-indexed variables, there is always exactly one instance of the variable that carries semantic information. In our baseline method, similar to Barzdins and Gosko (2016) and Konstas et al. (2017), we simply copy this semantic information, while removing the variables, as is shown below. This method does never output reentrancy nodes and therefore functions as a baseline. An example is shown in Figure 2.

```
(like-01                                (want-01
    :ARG0 (person :name "Bob")              :ARG1 (person :name "Jack")
    :ARG1 (person :name "Bob"))             :ARG3 (buy-01
                                                  :ARG1 (person :name "Jack")
                                                  :ARG2 (ice-cream)))
```

Figure 2: Baseline representations of examples (1) and (2).

Method 1B: Reentrancy Restoring This method is created to restore reentrancy nodes in the output of the baseline model. It operates on a very ad hoc principle: if two nodes have the same concept, the second one was actually a reference to the first one. We therefore replace each node that has already occurred in the AMR by the variable of the antecedent node. This approach was applied by van Noord and Bos (2017) and Foland and Martin (2017).

The model thus never *learns* to output reentrancy, but the co-indexed variables are restored in a post-processing step. An example is shown in Figure 3. Note that this process can also erroneously insert reentrancies when two separate entities would be correct. For example, if the sentence *Bob*$_i$ *likes Bob*$_j$ refers to two different Bobs, the initial AMR in Figure 3 would have been correct.

```
(l / like-01                              (l / like-01
  :ARG0 (p / person :name "Bob")            :ARG0 (p / person :name "Bob")
  :ARG1 (p2 / person :name "Bob"))          :ARG1 p)
```

Figure 3: Example of the Reentrancy Restoring method for example (1). Initial AMR on the left, reentrancy restored AMR on the right.

Method 2: Indexing This method comprises of removing all variables from an AMR, except when they are co-indexed. The remaining variables are normalised by converting them to numbers, so that each unique co-indexed variable has a unique identifier. Similar approaches were applied by Peng et al. (2017) and Buys and Blunsom (2017), and an example is shown in Figure 4. In this approach the model actually learns where it should output reentrancy nodes, instead of restoring them in a post-processing step.

However, we do still need some post-processing, as is shown in Figure 5 (after variables are restored). In this AMR, there is an index that is never instantiated (`*3*`) and an instantiated index that is never

[2]All pre- and post-processing scripts are available at https://github.com/RikVN/AMR.

```
(like-01                                      (want-01
  :ARG0 (*1* person :name "Bob")                  :ARG1 (*1* person :name "Jack")
  :ARG1 (*1*))                                       :ARG3 (buy-01
                                                         :ARG1 *1*
                                                         :ARG2 (ice-cream)))
```

Figure 4: Representations of examples (1) and (2) when applying the Indexing method.

referred to (*2* b / buy). The superfluous *2* can simply be removed, but for *3* we have multiple options. Algorithm 1 shows how we handle these cases.

For each referent that was never instantiated, we first check if there is also an instantiated index that is never referred to. If that it is the case, we replace it by that variable. This assumes that the model merely mismatched the index symbols. This works for *3* in Figure 5, even though the resulting node is still incorrect. If that was not the case, we try to replace the referent by an instantiated index that already did have a referent. Since this index already has a referent, we assume it is likely that a mismatched referent actually referred to this index (in Figure 5, *3* would then be replaced by p) . In the case that both previous options failed, we simply pick the variable of the concept that is most often a referent in the training set.

```
(w / want-01                                   (w / want-01
   :ARG1 (*1* p / person :name "Jack")            :ARG1 (p / person :name "Jack")
          :ARG3 (*2* b / buy-01                         :ARG3 (b / buy-01
                :ARG1 (*1*)                                   :ARG1 p
                :ARG2 (i / ice-cream)                         :ARG2 (i / ice-cream)
                :ARG3 (*3*)))                                 :ARG3 b))
```

Figure 5: Possible output of the model for example (2) on the left, while the fixed AMR after applying Algorithm 1 is shown on the right.

Algorithm 1 Post-processing algorithm used to replace indexes by variables.

R: all referent indexes in output (e.g. *1*, *3*)
X: all instantiated indexes that do not have a referent (e.g. *2* b / buy-01)
Y: all instantiated indexes that have a referent (e.g. *1* person :name "Jack")
C: all concept-variable pairs (e.g. w / want-01)
most_freq: function that, given a list of concepts, selects the concept that is most frequently a referent in the training set

```
for all ref in R do
    if ref $\subseteq$ X then
        replace ref by X(ref)
    else if X $\neq \emptyset$ then
        replace ref by most_freq(X)
    else if Y $\neq \emptyset$ then
        replace ref by most_freq(Y)
    else
        replace ref by most_freq(C)
    end if
end for
```

Method 3: Absolute Paths In this method reentrancy is established by replacing a co-indexed variable by an absolute path that describes a position within an AMR. Absolute paths start from the top (root) of the AMR. The examples in Figure 6 show the relative paths. In the first AMR, the path `:ARG0` describes that the node refers to the value of the relation `:ARG0`. In the second AMR, the path `:ARG1` refers in a similar way to `:ARG1`. Note that although these examples are straightforward, the paths can become quite long for larger AMRs, e.g. `:op1 :ARG0 :ARG0-of :mod`. The longest path in the training set even contains 14 relations.

```
(like-01
  :ARG0 (person :name "Bob")
  :ARG1 {:ARG0})
```

```
(want-01
  :ARG1 (person :name "Jack")
     :ARG3 (buy-01
          :ARG1 {:ARG1}
          :ARG2 (ice-cream)))
```

Figure 6: Representations of examples (1) and (2) when applying the Absolute Paths method.

This method is perhaps the most attractive from a theoretical point of view. Note, however, that not every node in an AMR can be described by a unique path, as ambiguities might occur when there are two edges with the same name (this can occur, for instance, when more than one modifier is present). To solve these ambiguities an index (e.g. |1|, |2|) is added to each relation in the path.[3] This was necessary for 632 out of the 40,582 paths in the training set. In total, there are 5,760 unique paths, of which 3,447 only occur once. The most frequent path is `:ARG0|1|`, occurring 5,405 times.

Similar to the Indexing method, a problem with this approach is the fact that we have no control over what the model will output. Especially for larger AMRs it is likely that the model will output impossible paths without destination. For example, the model might output `:ARG2` instead of `:ARG0` in the left example in Figure 6.

These impossible paths still need to be replaced by a variable to get a valid AMR. The strategy we opt for here is similar to one used in the Indexing method, namely replacing the path by the variable of the concept in the AMR that most frequently has a referent in the training set.

3.2 Neural model

We implement a bidirectional sequence-to-sequence model with general attention that takes characters as input, using the OpenNMT software (Klein et al., 2017). The parameter settings are the same as in van Noord and Bos (2017) and are shown in Table 2. It is trained for 20 epochs, after which the model that performs best on the development set is used to decode the test set.

Table 2: Parameter settings of the seq2seq model, as in van Noord and Bos (2017).

Parameter	**Value**	**Parameter**	**Value**
Layers	2	RNN type	brnn
Nodes	500	Dropout	0.3
Epochs	20	Vocabulary	100–200
Optimizer	sgd	Max length	750
Learning rate	0.1	Beam size	5
Decay	0.7	Replace unk	true

3.3 Experiments

We test the impact of the different methods on two of our earlier models, described in van Noord and Bos (2017). The first is a simple baseline model that only takes the characters into account without any

[3]For clarity, this was omitted in the example in Figure 6.

additional methods to improve performance. This model is referred to as the char-only model.

The second is the approach that produced one of the best results so far in the literature. This model uses POS-tagged input, clusters together groups of characters (super characters) and exploits 100,000 "silver" AMRs that were obtained by using the off-the-shelf AMR parsers CAMR (Wang et al., 2015) and JAMR (Flanigan et al., 2014)[4]. The added AMRs are all CAMR-produced. We must note that CAMR is not particularly keen on outputting coreference, as the 100,000 silver AMRs only produced 18,865 new reentrancy nodes.

The second approach also employs the postprocessing methods Wikification and pruning, as explained in van Noord and Bos (2017). The Wikification step simply adds wiki links to `:name` nodes, since those links were removed in the input. Pruning is used to remove erroneously produced duplicate output. This is a common problem for sequence-to-sequence models, since the model does not keep track of what it has already output. No pre-training or ensemble methods are used for both approaches.

4 Results and Discussion

4.1 Main results

The results of applying our three methods on the baseline and best model are shown in Table 3. All reported numbers are F-scores obtained by using SMATCH (Cai and Knight, 2013). All three methods offer an improvement over the baseline, for both the baseline and the best model. Indexing is the highest scoring method, except for the test set of the best model, since Reentrancy Restoring obtains the same F-score there. Explicitly encoding the absolute paths resulted in an increase over the baseline, but did not outperform both Reentrancy Restoring and Indexing, although only by a small margin.

Table 3: Results of the different methods in comparison to the char-only and best model.

	Char-only		**Best model**	
	Dev	**Test**	**Dev**	**Test**
Baseline	54.8	53.1	69.3	68.0
Reentrancy Restoring	55.7	54.2	70.0	69.0
Indexing	55.9	56.0	70.5	69.0
Absolute Paths	54.9	53.9	70.3	68.7

Table 4: Results for each method on AMRs with (coref) and without (no coref) reentrancy nodes.

	Char-only				**Best model**			
	Coref		**No coref**		**Coref**		**No Coref**	
	Dev	**Test**	**Dev**	**Test**	**Dev**	**Test**	**Dev**	**Test**
Baseline	52.1	50.3	61.6	60.6	66.9	65.1	73.8	74.8
Reentrancy Restoring	53.4	52.0	61.2	60.8	68.3	66.8	73.8	74.8
Indexing	54.2	53.5	60.5	61.6	68.6	66.5	74.8	74.8
Absolute Paths	52.8	51.5	59.6	59.2	68.2	66.3	74.6	74.5

It is interesting to look at whether we indeed improve on parsing AMRs that have co-indexed variables, in comparison to AMRs that do not contain them. This is shown in Table 4. We see that, in general, we indeed only improve on both baselines for AMRs that have co-indexed variables. This is the case for both the dev and the test set. This is the desired scenario: we improve on AMRs with reentrancy, while

[4]Please see van Noord and Bos (2017) for a full explanation of this process.

performance does not decrease on AMRs without reentrancy. Only applying the Absolute Paths method on the baseline model scores lower on the test set on AMRs without reentrancy nodes. Similar to the results in Table 3, Indexing outperforms Reentrancy Restoring for the baseline model, but has similar scores when applied to the best model.

4.2 Detailed analysis

Table 5 shows the how often the model outputs possible paths for the Absolute Paths method. Unfortunately, about 50% of the time, the model output a path that did not lead to a possible referent, leaving us to rely on the frequency heuristic. This problem is not as severe for our best model, though, since about 75% of paths were actually possible. The addition of extra (silver) data thus helped the model in learning the paths, suggesting that the baseline results might merely be an effect of sparse data. However, the very long paths still proved to be challenging for the best model. This might mean that, even when adding more gold data, learning such sophisticated structures is too difficult for end-to-end sequence-to-sequence models in general.

Table 5: Number of possible (pos) and impossible (imp) paths in the output, for the dev and test set for both our models, when using the Absolute Paths method.

	Char-only				**Best model**			
	Dev		Test		Dev		Test	
	Pos	Imp	Pos	Imp	Pos	Imp	Pos	Imp
Total paths produced	455	464	534	533	636	200	783	328
Paths of length 1	160	30	148	32	231	12	193	13
Paths of length 2	211	202	265	194	268	68	398	108
Paths of length 3	76	156	106	210	130	80	167	123
Paths of length >3	8	76	15	97	7	40	25	84

Table 6 shows more detailed results of the Indexing method. In the majority of cases, the model does what it is supposed to do: first instantiating an index, then referring to it. However, for finding the correct variable replacement for the baseline model, we still have to fall back on heuristics in approximately 30% of the cases. Most of these instances are then solved by referring to the concept that is most frequently a referent in the training set. However, for our best model, this reliance is not there anymore. The model generally only outputs an index when it was instantiated first, which is exactly what was intended.

Table 6: Number of times each step in Algorithm 1 was responsible for replacing an index in the output when using the Indexing method.

	Char-only		**Best model**	
Replace index by	**# Dev**	**# Test**	**# Dev**	**# Test**
Variable that was instantiated	749	984	770	1,122
Variable that was instantiated but not referred to	25	28	0	0
Variable of most frequent instantiated index	8	8	3	9
Variable of most frequent concept	248	217	3	10

5 Conclusion

We proposed three methods to handle co-indexed variables for neural semantic (AMR) parsing. The best results were obtained by the Indexing method, which explicitly encodes co-indexing nodes in the data set. The perhaps theoretically most attractive Absolute Paths performed the worst, although it did still offer an improvement over the baseline. Perhaps an interesting direction for future research is to use relative instead of absolute paths, encoding the path relative to the reentrancy node instead of starting at the top of the AMR, because this will make the paths shorter and more local.

Acknowledgements

We thank the two anonymous reviewers for their comments. We also used a Tesla K40 GPU, which was kindly donated to us by the NVIDIA Corporation. We would also like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster. This work was funded by the NWO-VICI grant "Lost in Translation — Found in Meaning" (288-89-003).

References

Banarescu, L., C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider (2013, August). Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, Sofia, Bulgaria, pp. 178–186. Association for Computational Linguistics.

Barzdins, G. and D. Gosko (2016, June). Riga at semeval-2016 task 8: Impact of smatch extensions and character-level neural translation on amr parsing accuracy. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, San Diego, California. Association for Computational Linguistics.

Buys, J. and P. Blunsom (2017, July). Robust incremental neural semantic graph parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vancouver, Canada, pp. 1215–1226. Association for Computational Linguistics.

Cai, S. and K. Knight (2013, August). Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Sofia, Bulgaria, pp. 748–752. Association for Computational Linguistics.

Damonte, M., S. B. Cohen, and G. Satta (2017, April). 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*, Valencia, Spain, pp. 536–546. Association for Computational Linguistics.

Flanigan, J., S. Thomson, J. Carbonell, C. Dyer, and N. A. Smith (2014, June). A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Baltimore, Maryland, pp. 1426–1436. Association for Computational Linguistics.

Foland, W. and J. H. Martin (2017, July). Abstract meaning representation parsing using lstm recurrent neural networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vancouver, Canada, pp. 463–472. Association for Computational Linguistics.

Klein, G., Y. Kim, Y. Deng, J. Senellart, and A. Rush (2017, July). Opennmt: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, Vancouver, Canada, pp. 67–72. Association for Computational Linguistics.

Konstas, I., S. Iyer, M. Yatskar, Y. Choi, and L. Zettlemoyer (2017). Neural amr: Sequence-to-sequence models for parsing and generation. *arXiv preprint (accepted in ACL-2017) arXiv:1704.08381*.

Peng, X., C. Wang, D. Gildea, and N. Xue (2017, April). Addressing the data sparsity issue in neural amr parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, Valencia, Spain, pp. 366–375. Association for Computational Linguistics.

Pust, M., U. Hermjakob, K. Knight, D. Marcu, and J. May (2015). Parsing english into abstract meaning representation using syntax-based machine translation. *Training 10*, 218–021.

van Noord, R. and J. Bos (2017). Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *Accepted in CLiN journal, arXiv preprint arXiv:1705.09980*.

Wang, C., N. Xue, and S. Pradhan (2015, May–June). A transition-based algorithm for amr parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Denver, Colorado, pp. 366–375. Association for Computational Linguistics.