Bálint Sass

# THE "DEPENDENCY TREE FRAGMENTS" MODEL FOR QUERYING A CONSTRUCTICON

**Abstract** In this paper, we argue that it is beneficial for a constructicon to have a query interface where the user can enter arbitrary text. To allow this, we present a dependency-tree-based model for representing constructions, and show that this model can serve as a basis for working out a user-friendly query interface which analyses the free-text user query and matches the constructicon entries to it, revealing all constructions from the query text for the user, without expecting any knowledge of construction grammar.

**Keywords** constructicon; construction; dependency tree; dependency tree fragments; Universal Dependencies; Hungarian

## 1. Motivation: The User's Perspective

We consider inventories of constructions (i.e., constructicons) as a replacement for dictionaries, not as a supplement to them. We agree with Hilpert (2014, p. 2) as he states that "a person's knowledge of language consists of nothing but constructions". Also agreeing with Goldberg (2006) who includes bound and free morphemes as well into the set of cxns[1] and contradicting Haspelmath (2023) who excludes items without open slots, we consider every linguistic form having a meaning a cxn regardless of how complicated or how simple it is. This way, a ccn covers everything a dictionary does and much more.

The basic way of interaction with a dictionary is to provide an input word and obtain its dictionary entry. In the past, you were not able to do this without knowing some kind of canonical form of the word. Traditional dictionaries have had cross-referencing to help finding the canonical form corresponding to an irregular form, while modern dictionaries do this for all regular forms as well (look up e.g., *books* in OALD (Oxford University Press, 2023)). We think that a ccn should have a similar functionality extended to possible multiword inputs: it should be able to return the (canonical form of) cxns present in the input text regardless of the specific form they have.

As a kind of dictionary, a ccn is intended for the public: mainly for language learners and all those who are interested. We do not want it to be "probably best suited for users with better than average knowledge of linguistics" (cf. Lyngfelt et al., 2018b, p. 92). We do not expect the user to know some canonical form of the cxn in question not even the theoretical notion of what a cxn is. Note that the canonical form of cxns may be much more complex than a canonical form of words. It is some kind of machine-readable formal representation handling different kinds of morphemes, connections, slots and fillers.

[1] To improve readability, we use the abbreviation *ccn* for constructicon and *cxn* for construction.

We expect the user to have a (multiword) linguistic utterance in mind, and that he/she wants to uncover its structure and meaning. We think that the solution is the following. To function similarly to a dictionary a ccn should have a dynamic machinery in the background: it should be able to accept arbitrary short text, analyse it on the fly and reveal the cxns in it to the user. For example, it should reveal the cxn '*take something for granted'* for inputs like *I take it for granted*, or even *some French journalist actually took this story for granted*, or *it was taken for granted*.

Our proposed solution can be compared to the following approaches: (1) to present a browsable list of cxns to the user; (2) to work out a sophisticated query interface which reflects the subtleties of the cxn-representation; (3) to give the user a formal query language which works on the cxn-representation. Ccns tend to use the first two approaches (cf. the Swedish ccn (Lyngfelt et al., 2018b) and the Russian ccn (Janda et al., 2020; Bast et al., 2021)). None of these three approaches are ideal in our opinion. The first approach is too limiting in two ways: firstly, it is something like browsing pre-created answers instead of having the possibility to ask a question, secondly, allowing to look at just one cxn at a time it does not make it possible to investigate the interconnections of cxns. The other two approaches expect some knowledge of the cxn-representation which we would like to hide from the user as we argued above.

To work out our approach we need (1) a representation which is capable to represent any kind of cxns; (2) to store the entries of the ccn in this form; (3) an algorithm to analyse the input text according to the representation; and (4) another algorithm to match the ccn-entries to the representation of the input query. To elaborate this representation, the starting idea is to use dependency trees with filled vertices and empty vertices (corresponding to open slots) as well to represent cxns.

## 2. The Hungarian Constructicon

We are implementing the above approach for the Hungarian ccn which is in preparation.

The Hungarian Constructicon project (Sass, 2023) follows the approach of processing an existing dictionary and extracting cxns from it to create the database of the ccn. Not just headwords become cxns, but also expressions that are hidden in the "expressions" part and even in the "examples" part of the entries. All of these become ccn-entries (or head-constructions (hcxns)) of their own, together with their definitions. For example, Hungarian counterparts of *action* and *take part in* will be hcxns of the same right. After the above processing, what is listed as hcxn in the ccn is accepted as a cxn. In other words, we leave the burden of defining cxns to the original dictionary.

## 3. The "Dependency Tree Fragments" Model

Based on the first section, we can consider a concrete single morpheme (Goldberg, 2006, p. 5) a cxn, and at the other end of the scale we can consider an abstract grammatical rule (a constructional schema (Diessel, 2023, p. 16)) a cxn as well. Cxns consist of two kinds of elements: fixed elements (fillers) and open slots where other cxns may be inserted as a filler (Lyngfelt et al., 2018a). Taking the above examples, a single morpheme is a fixed element in itself, and a grammatical rule contains only open slots. The main advantage of the constructionist approach is precisely that it can handle all of the above in a unified framework as cxns.

Most interesting cxns are in between, these are the cxns that have both kinds of elements: for example, *take part in*. Beyond its three fixed elements, this cxn has two open slots as well, one for the subject and one for the *in*-PP. This cxn is complete only together with all of these, it should be included in a ccn as an entry in its complete form. We can think of these mixed expressions as the genuine building blocks of language, of which, words and grammatical rules are only special cases.

Building on the concept of “dependency tree fragments” (e.g., Morimoto et al., 2016), our basic idea is the following: dependency-tree-like structures can represent not just texts but cxns as well. The main difference to handle is this: while in texts all slots are filled with some words, in cxns some slots are not filled. The essential point is that we can represent open slots appropriately, specifically, by edges having no vertex at the bottom end of the edge (Figure 1).

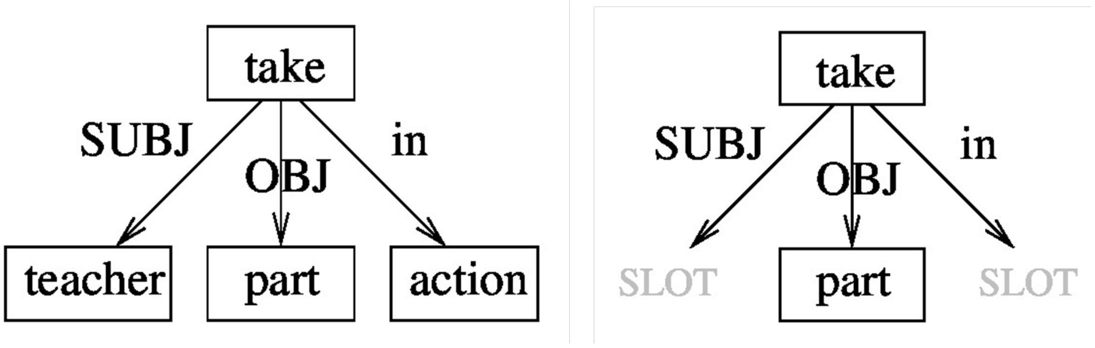

**Fig. 1:** Left side: representation of text *the teacher takes part in the action*. Right side: representation of cxn *take part in* together with its two open slots. Open slots are represented by edges having no vertex at the bottom end.

As an alternative to drawing trees (cf. Figure 1) we can use the following equivalent textual representation respectively:

```
[take /SUBJ:teacher /OBJ:part /in:action]

[take /SUBJ:_ /OBJ:part /in:_]
```

An edge is marked by `[/SLOT]`, a vertex is marked by `[:filler]`, and the notation for an open slot is `[:_]` (underscore).

As our implementation is for Hungarian, we will use Hungarian examples in the remaining part of this paper.

## 4. The Same Analysis for Constructicon Entries and Query Texts

In order to be able to compare the free-text queries with our ccn-entries we use the same representation formalism – namely the "dependency tree fragments" model described in Section 3 – for both of them.

For creating the representation, we utilize a classical natural language processing pipeline. For tokenization, morphological analysis and POS-tagging, we use the `emtsv` system (Indig et al., 2019; Mittelholcz, 2017; Novák et al., 2016; Orosz & Novák, 2013) combined with the dependency-analysis module of the `udpipe` system (Straka et al., 2016) which is integrated into `emtsv`, and provides an output which meets the Universal Dependencies standard.

Free-text queries are obviously texts. Initially, ccn-entries has a simple textual form as well, as they come from a dictionary. The important difference that hcxns have open slots, is somewhat hidden in this form in an unformalized way. A ccn-entry can be the following, for example: *'részt vesz valamiben'* ('take part in something'). What we want to emphasize here is what appears as *'valami'* ('something') in the textual form of a hcxn is in fact an open slot in the vast majority of cases. Our representation allows us to formalize this information, so after running the analysis we transform it to an open slot on the fly. It is important that although *'valami'* is deleted, its suffix *'ben'* is retained. This is a Hungarian inessive (INE) case marker with a meaning similar to the English preposition *in*. Clearly, it should be kept, as it marks the open slot in the cxn in question. This is how the ccn-entry *'részt vesz valamiben'* becomes `[vesz /OBJ:rész /INE:valami]` after analysis and then `[vesz /OBJ:rész /INE:_]` after the open slot transformation. The latter form is stored in the ccn database alongside the original textual form of the ccn-entry.

Concerning slots around verbs in Universal Dependencies, beyond `nsubj` (subject), `obj` (object) and `iobj` (indirect object), we have `nmod:obl` (oblique) among the most frequent ones. The latter category is too coarse-grained for our purposes. We decided to retain the case of the oblique as well, in other words we break down the oblique category by case, because the case carries essential information about the slot.

Having run the analysis on all ccn-entries we obtain a ccn augmented with a formal representation for each hcxn. Also, we have the same algorithm at hand to provide formal representation for free-text input queries.

## 5. An Algorithm for Identification of cxns in Query Text

Our goal is to extract hcxns from free-text user queries. Fixed, continuous cxns are easier to handle by some string-matching-based methods, but Hungarian verbal cxns pose a more difficult task, as they can be non-continuous and can be in different word orders. The proposed representation helps us by abstracting from these properties.

The main algorithm of our proposed system is intended for identifying cxns in free-text queries. The big picture is the following: (1) the user enters a query; (2) the system assigns "dependency tree fragments" representation to it by automatic linguistic analysis; (3) then compares the query against the ccn-entries in our ccn database which have the same representation; (4) recognizes cxns in the query; (5) and presents them to the user finally.

A query can contain several cxns. In the sentence *'a tanár részt vesz az akcióban'* ('the teacher takes part in the action') the main cxn is clearly *'SLOT$_{SUBJ}$ részt vesz SLOT$_{INE}$'* ('SLOT$_{SUBJ}$ take part SLOT$_{in}$'). But as words are cxns as well, there are others: *'tanár'* ('teacher') and *'akció'* ('action') not to mention the two article-cxns. We would like to show all of them to the user.

Our most important observation is the following. Notice that all cxns are a specific part of the representation tree of the query, and on the other hand, these specific parts add up to the tree without remaining fragments. So our task is to "split up" the tree into cxns somehow. An illustration can be seen in Figure 2.

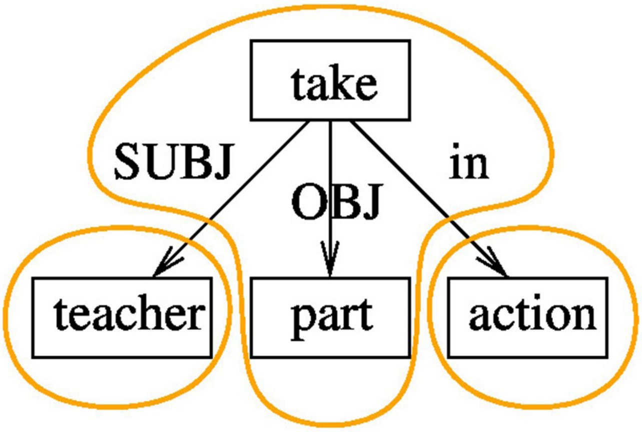

**Fig. 2:** The three cxns into which the sentence *'a tanár részt vesz az akcióban'* ('the teacher takes part in the action') is split up by our "dependency tree fragments" algorithm. The two article-cxns are not depicted.

We propose the following algorithm.

1. start from the root vertex of the tree of the query text;
2. taking the ccn, search for hcxns which *matches* here;
3. take the longest matching hcxn if there are several;
4. *remove* the found hcxn from the tree;
5. after the removal the tree falls apart into smaller trees;
6. take these smaller trees one by one and start over the algorithm.

In step 2 above, "*H* (a hcxn) *matches T* (query text)" means that the root is the same in their representation, *T* contains all the direct edges branching from the root vertex what *H* contains, and the vertices at the end of these edges are the same or *H* has an open slot there. In other words, an open slot in the hcxn matches any fillers present in the query. If *H* has further vertices and edges, they should match similarly in a recursive manner. Basically, this is something like determining whether *H* is in a subset-like relation with *T*.

Performing *remove* (in step 4 above), if the cxn to remove has filled slots at its lowest level, we keep these vertices in order to being able to retain the edges starting from them.

If we get to a single vertex/filler (e.g., *'tanár'* ('teacher') in Figure 2), we apply morphological analysis to split up the word into a series of morphemes. Then we attempt to match the entries of the ccn onto this series. This is a similar procedure what we do with trees, but much simpler, as the morpheme structure of a word is not a tree but a simple chain.

The reader can follow the operation of the algorithm in Figure 2. The first step of splitting up the sentence into cxns is matching the hcxn on the right side of Figure 1 onto the original text. After removing this hcxn, what remains is two one-vertex tree, each of them representing a monomorphemic filler.

## 6. Examples

The example we have analysed throughout the paper in Figures 1 and 2 had the following structure: at root there was a multi-element cxn with two slots, and the fillers in both of the slots were simple single-element fillers. Of course, the algorithm works well on inputs of a different structure as well. We illustrate this through two more examples.

A single-element root hcxn can have a multi-element filler as we see in the example *'megérkezik az alapító tag'* ('the founding member arrives'). The representation of the query is `[megérkezik /SUBJ:tag /SUBJ/amod-att:alapító /SUBJ/det:az]`. We note that `[/V1/V2:F]` notation means a two-edge path in the tree from the root: a `[V1]` vertex, then a `[V2]` vertex connected to the end of `[V1]`, and `[V2]`'s own filler is `[F]`. After processing the root element as a single word we get the following representation: `[tag /amod-att:alapító /det:az]`. We have *'alapító tag'* ('founding member') as a hcxn with the representation `[tag /amod-att:alapító]` in the ccn. We can see that this will match because the query fragment being processed contains all the edges that it has, and there is no contradiction between fillers. At the end we will have three cxns: *'megérkezik'* ('arrive'), *'alapító tag'* ('founding member') and an article-cxn.

Of course there can be more than one multi-element hcxn in the query text. In the case of *'az alapító tag részt vesz az akcióban'* ('the founding member takes part in the action') not just the root element, but one of the fillers is a multi-element hcxn. The algorithm handles this similarly to the above. As we see, cxns occur as fragments of dependency trees as the method's name suggests.

It is worth to mention that the algorithm implements a kind of "fallback" mechanism. First at the level of words, then at the level of morphemes. If no multi-element hcxns are found at the current root vertex, then the algorithm falls back to process the vertices (fillers, words) one by one; and then if no single-element hcxn is found at the current root vertex, the filler’s morphemes are processed one by one. As an example for the latter, *‘eseményekről’* (‘about events’) consists of three morhpemes which are three separate cxns: *‘esemény’* (‘event’), *‘-ek’* (plural marker) and *‘-ről’* (‘about’).

## 7. Demo Implementation

The main contribution of this paper is the model itself together with the two algorithms operating it described above. In order to give the reader an impression of how this might work in practice we created a demo implementation as well. It has a limited feature set, its main purpose is to demonstrate the essential properties of the model, the basic operation and feasibility of the proposed system and show its potentials. To create a fully operational system is a long term goal.

The “dependency tree fragments” demo is available here as part of the Hungarian Constructicon project: https://ccn.nytud.hu (username: `demo`, password: `letssee`). You can see how the system works for examples mentioned in the paper, and also, you can test the system entering other similar short Hungarian texts.

## References

Bast, R., Endresen, A., Janda, L. A., Lund, M., Lyashevskaya, O., Mordashova, D., Nesset, T., Rakhilina, E., Tyers, F. M., & Zhukova, V. (2021). The Russian Constructicon. An electronic database of the Russian grammatical constructions. https://constructicon.github.io/russian

Diessel, H. (2023). *The Constructicon.* Cambridge University Press.

Goldberg, A. E. (2006). *Constructions at work: The nature of generalization in language.* Oxford University Press.

Haspelmath, M. (2023). On what a construction is. *Constructions, 15*(1), 1–15. doi 10.24338/cons-539

Hilpert, M. (2014). *Construction Grammar and Its Application to English.* Edinburgh University Press.

Indig, B., Sass, B., Simon, E., Mittelholcz, I., Vadász, N., & Makrai, M. (2019). One format to rule them all – The emtsv pipeline for Hungarian. In A. Friedrich, D. Zeyrek, & J. Hoek (Eds.), *Proceedings of the 13th Linguistic Annotation Workshop* (pp. 155–165). Association for Computational Linguistics.

Janda, L., Endresen, A., Zhukova, V., Mordashova, D., & Rakhilina, E. (2020). How to build a constructicon in five years: The Russian example. *Belgian Journal of Linguistics, 34*, 162–175.

Lyngfelt, B., Borin, L., Ohara, K., & Torrent, T. T. (Eds.). (2018a). *Constructicography: Constructicon development across languages.* John Benjamins.

Lyngfelt, B., Bäckström, L., Borin, L., Ehrlemark, A., & Rydstedt, R. (2018b). Constructicography at work: Theory meets practice in the Swedish Constructicon. In B. Lyngfelt, L. Borin, K. Ohara, & T. Torrent (Eds.), *Constructicography: Constructicon development across languages* (pp. 41–106). John Benjamins.

Mittelholcz, I. (2017). emToken: Unicode-képes tokenizáló magyar nyelvre. [emToken: a Unicode-capable tokenizer for Hungarian.]. In V. Vincze (Ed.), *Proceedings of MSZNY2017* (pp. 70–78). Szegedi Tudományegyetem, Informatikai Tanszékcsoport.

Morimoto, A., Yoshimoto, A., Kato, A., Shindo, H., & Matsumoto, Y. (2016). Identification of Flexible Multiword Expressions with the Help of Dependency Structure Annotation. In E. Hajičová, & I. Boguslavsky (Eds.), *Proceedings of the Workshop on Grammar and Lexicon: interactions and interfaces (GramLex)* (pp. 102–109). The COLING 2016 Organizing Committee.

Novák, A., Siklósi, B., & Oravecz, Cs. (2016). A New Integrated Open-source Morphological Analyzer for Hungarian. In N. Calzolari, K. Choukri, T. Declerck, S. Goggi, M. Grobelnik, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, & S. Piperidis (Eds.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. European Language Resources Association (ELRA).

Orosz, G., & Novák, A. (2013). PurePos 2.0: a hybrid tool for morphological disambiguation. In R. Mitkov, G. Angelova, & K. Bontcheva (Eds.), *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2013)* (pp. 539–545). INCOMA Ltd. Shoumen.

Oxford University Press (2023). Oxford Advanced Learner's Dictionary. https://www.oxfordlearnersdictionaries.com

Sass, B. (2023). From a dictionary towards the Hungarian Constructicon. In M. Medveď, M. Měchura, C.Tiberius, I. Kosem, J. Kallas, M. Jakubíček, & S. Krek (Eds.), *Electronic lexicography in the 21st century (eLex 2023): Invisible Lexicography. Proceedings of the eLex 2023 conference* (pp. 534–544). Lexical Computing CZ s.r.o.

Straka, M., Hajič, J., & Straková, J. (2016). UDPipe: Trainable Pipeline for Processing CoNLL-U Files Performing Tokenization, Morphological Analysis, POS Tagging and Parsing. In N. Calzolari, K. Choukri, T. Declerck, S. Goggi, M. Grobelnik, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, & S. Piperidis (Eds.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)* (pp. 4290–4297). European Language Resources Association (ELRA).

## Acknowledgements

The research described in this paper was supported by the OTKA (K 147452) grant of the National Research, Development and Innovation Office (NKFIH).

## Contact information

**Bálint Sass**
HUN-REN Hungarian Research Centre for Linguistics, Institute for Lexicology
sass.balint@nytud.hu