A Declarative Formalism for Constituent-to-Dependency Conversion

Torsten Marek Gerold Schneider Martin Volk

Institute of Computational Linguistics
University of Zürich

marek@ifi.uzh.ch gschneid@ifi.uzh.ch volk@cl.uzh.ch

Abstract

In this paper, we present a declarative formalism for writing rule sets to convert constituent trees into dependency graphs. The formalism is designed to be independent of the annotation scheme and provides a highly task-related syntax, abstracting away from the underlying graph data structures.

We have implemented the formalism in our search tool and used a preliminary version to create a rule set that converts more than 97% of the TIGER corpus.

1 Introduction

Syntactic structures are expressed either as constituent or as dependent structures. The fundamental differences between the two formalisms is that dependency is (1) endocentric, which means that the category of the parent node follows from the category of the child node, and (2) strictly binary, which means that the conversion of ternary rules (and n-ary if n > 2) is ambiguous. Debates about which formalism is theoretically more appropriate for linguistics are no longer topical, [4] has e.g. shown that X-bar grammar (Principles and Parameters) is equivalent to Dependency, but the practical problem of conversion persists. On the one hand, the theoretical discussions have led to successful mixed formalisms (e.g. HPSG, LFG). On the other hand, tackling the intricate details of a given annotation scheme is now more seen as a practical problem, where the devil is in the details. We present a declarative formalism and a tool that supports the linguist in writing conversion rules. The tool graphically displays conversions sentence-by-sentence and immediately updates rule changes. It also delivers errors and coverage reports, allowing a linguist to write a broad-coverage conversion in little time.

The paper is structured as follows: We summarize related work in section 2. The conversion formalism is introduced and illustrated with examples in section 3. We describe the performance of a sample conversion rule set that we have written for the German TIGER Treebank in section 4. We discuss future work in section 5.

2 Related Work

On an abstract level, conversion of a constituent tree into a dependency graph is an application of *graph transformation* [8], which is used in fields like compiler theory and software architecture validation. A graph transformation is a series of rewrite rules which are applied to a source graph to create a target graph. Our formalism is influenced by this idea, but while graph transformations are domain-oblivious, our formalism is coupled strongly to the domain of syntactic annotation as graphs. On a practical level, a number of conversion algorithms already exist.

2.1 MALTparse

[12] for example maps Penn Treebank constituents to dependency representations. As the majority of Penn Treebank constituents do not have functional labels, relation labels need to be assigned. In order of descending priority, the rules are as follows. (*mother = M, head = H, dependent = D, r = dependency relation label*)

1. if *D* is a punctuation category, *r* = P.
2. if *D* contains the function tag SBJ, *r* = SBJ.
3. if *D* contains the function tag PRD, *r* = PRD.
4. if *M* = *VP*, *H* is a part-of-speech tag and *D* = NP (without any function tag), *r* = OBJ.
5. if *M* = *VP*, *H* is a part-of-speech tag and *D* = VP, *r* = VC.
6. if *M* = *SBAR* and *D* = S, *r* = SBAR.
7. if *M* = *VP, S, SQ, SINV or SBAR*, *r* = VMOD.
8. if *M* = *NP, NAC, NX or WHNP*, *r* = NMOD.
9. if *M* = *ADJP, ADVP, QP, WHADJP or WHADVP*, *r* = AMOD.
10. if *M* = *PP or WHPP*, *r* = PMOD.
11. Otherwise, *r* = DEP.

This mapping is simple and reliable but leads to a less elaborate dependency formalism than e.g. the Stanford scheme [5]

2.2 Johannsson

[9] describes constituent-to-dependent conversion as a two stage process: head-selection (as most constituent representations are not endocentric) and function assignment (as most dependency relations are labeled). Head selection typically involves a set of Magerman rules [11]. The function assignment function is an extension of [12], addressing e.g. the distinction between object and adjunct, and raising long-distance dependency nodes.

2.3 Pro3Gres

[15] describes mapping as a complex task, which comes with some loss. In order to train the dependency parser Pro3Gres on Penn Treebank data, an involved mapping that has high precision but slightly incomplete recall, and that does not map all structural configurations is used. This leads to a mapping that delivers reliable relations but not fully connected trees, which is sufficient for the task of delivering statistical data for parser training. The annotation scheme is very similar to the Stanford scheme [7].

The structural configurations are queried with a large collection of tgrep queries, the majority of which are non-local, allowing access to lexical material and dealing with long-distance dependencies.

2.4 TiGerDB

There are several approaches converting the Penn Treebank, but there is less research on other Treebanks. Forst et. al [6] converted the German TIGER Treebank into a dependency representation, TiGerDB. The conversion had to be done semi-automatically, since TiGerDB has a richer annotation than the original data. Boyd et al. [1] address several issues in TiGerDB, which keep it from becoming a proper gold standard for dependency parsing and map it to a more surface-oriented analysis which does not include abstract nodes and is aligned with the original corpus tokens.

3 The Conversion Formalism

Developing a formalism for converting phrase structure into dependency trees that is independent of the annotation formalism, must strike a balance between expressivity on the one and task-relatedness on the other hand. If the formalism is too expressive, it might just as well be implemented as a framework or library for a general purpose programming language.

On the other hand, the formalism should have sufficiently expressive power, even for structures that can be arbitrarily large, like coordination constructions; and a syntax that captures the problems in constituent-to-dependency conversion, head identification and introduction of dependency edges.

3.1 Data Structures

In the conversion process, syntax trees, irrespective of the actual syntax formalism, are represented as directed graphs with typed nodes and edges. A graph G has a set of vertices (or nodes) V_G and a set of directed edges E_G. An edge going from p to c with $\{p,c\} \subset V_G$ is represented by $(p,c) \in E_G$.

Nodes and edges are typed attribute-value matrices (AVMs), with the actual types depending on the syntax and annotation formalism. There are several different types of edges, thus graphs need not be proper trees (i.e. one and only one node from which all others can be reached, no cycles), however the edges that define the *main structural layer* must satisfy the requirements for a tree.

3.1.1 Constituent Graphs

Based on the type system in [10], a constituent graph C consists of terminal and nonterminal nodes, representing words and linguistic phrases. The base type of all node types is the *feature record*. The main structural layer is defined by dominance edges, which define the phrase structure. All nodes but the root node have exactly one incoming dominance edge. If we refer to the origin and target node of a dominance edge, we use the terms *parent* and *child*.

The linear ordering of terminals in the sentence is annotated explicitly, which allows crossing dominance edges. This can be encoded using precedence edges between successive terminals, but for performance reasons each terminal has its positional index as a feature value.

3.1.2 Dependency Graphs

In contrast to constituent graphs, which are created from two different kinds of node types, a dependency graph D as defined in this paper contains only nodes representing words, again with explicit linear ordering.

The main structural layer is defined by dependency edges. The origin of a such an edge is called *head*, its target *dependent*. A word can have any number of outgoing dependency edges, but has at most one incoming dependency edge. Words without incoming edges are the root node and, depending on the formalism, punctuation and other non-word tokens.

Some formalisms for dependency graphs allow empty nodes, i.e. nodes that act as heads but do not correspond to any word in the sentence, thus resembling phrase nodes in constituent trees. In our conversions, we do not use empty nodes so far.

3.2 Conversion Process Constraints

Following the approach described in [9], conversion is carried out by identifying a head child for each phrase in the constituent tree and connecting the remaining children to the head using appropriate dependency edges. The conversion of a whole tree is carried out by repeated applications of *bound rules*, which define the rules for different kinds of linguistic phrases. The conversion process is *local* in the sense that only a phrase and its immediate children are considered during rule application. Nonterminal children are assumed to be opaque regarding their internal structure.

The conversion of a constituent graph C is *sound* if the generated dependency tree D satisfies the structural requirements described in section 3.1. The conversion is *complete* if each nonterminal node p is matched by a bound rule and if this rule handles each dominance edge emanating from p explicitly in the conversion process. The completeness criterion does not require that each dominance edge in E_c is converted into a dependency edge in E_D, cf. 3.3.2 for a justification of this relaxation.

Both constituent and dependency trees build structures on the exact same sentence material, thus the first step in the conversion is always to copy all terminal nodes from the constituent tree into the dependency tree.

3.3 Bound Rules for Nonterminals

Bound rules are the core of each rule set. They describe the conversion of a nonterminal node p and all its immediate children, i.e. all nodes $c \in V_C$ for which $(p,c) \in E_C$ holds. Example 1 shows the structure of such a rule.

(1) `context { <node> } { <body> }`

The keyword `context` is followed by the rule context `<node>`, which specifies the nonterminal nodes the rule is applied to. Here, we use the well-established syntax introduced by the TIGER query language [10], which can be used to query arbitrary acyclic directed graphs with feature structures as nodes. The context is currently limited to *node descriptions* and can thus only be used to select nodes based on local features. Extended contexts with structural constraints are discussed in section 5.1. The following list shows examples of node descriptions.

- `[cat="NP"]`
  Matches all nodes where the feature *cat* is `"NP"`
- `[cat=("CS"|"CNP")]`
  Matches all nodes where the feature *cat* is either `"CS"` or `"CNP"`

The second pair of curly braces contains the *body* (`<body>`) of a bound rule. The body is a series of actions, which are used to create the structure of the dependency graph, cf. section 3.3.2. They are applied to node variables which are introduced through quantifiers. The general form of a quantified expression is shown in example 2.

(2) `<quant> <var> { <child> } => <actions>;`

This expression states that the list of children of the current nonterminal node should be searched for nodes c that satisfy the local context given in `<child>`. The context is defined by a partial node description, which can be made more specific by constraining the label of the edge (p,c). Examples of child contexts are:

- `NK [pos="NN"]`
  Any word with part-of-speech tag `NN` and incoming dominance edge label `NK`.
- `AC [FREC]`
  Any node with incoming dominance edge label AC.

3.3.1 Quantifiers

The quantifier `<quant>` determines the way the individual nodes that satisfy the context descriptions are actually bound to the variable `<var>`. For the quantifiers, we follow the *Repeatability* criterion of [9] (p. 35), which states:

> Any syntactic relation must be either unlimitedly repeatable or non-repeatable.

The criterion marks the difference between complements, which must occur exactly once, and adjuncts, which can occur any number of times, including not at all, motivating the following quantifiers:

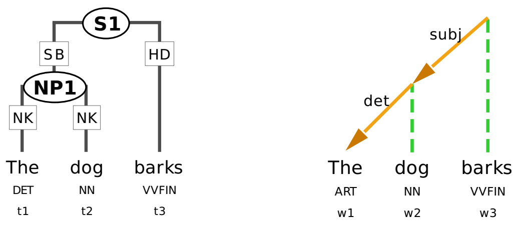
- **first**: Applies the actions only to the leftmost node that matches the criteria. The actions are mandatory; if they cannot be applied, the conversion fails immediately.
- **last**: This quantifier is analog to **first**, but matches the right- instead of the leftmost node, which is useful for right-headed phrases.
- **first?**, **last?**: These two quantifiers behave like their non-?-suffixed versions, except that they do not enforce application.
- **each**: Applies the actions to any node matching the criteria.

3.3.2 Actions

While the quantifiers are used to identify nodes, actions are used to describe the structure which is created. Examples for the two most important actions, head marking and dependency edge insertion, are shown in example 3. The quantified actions are applied from top to bottom.

(3)
```
context { #s:[cat="S"] } {
    first #hd { HD [FREC] } => root #hd;
    first #s { SB [FREC] } => <root> subj #s;
}
context { #np:[cat="NP"] } {
    last #n { NK [pos="NN"] } => root #n;
    each #d { NK [pos="ART"] } => <root> det #d;
}
```

The `root #hd` action marks its node arguments as the head of the current nonterminal, in this case the first (and only the first) occurrence of a node with the edge label `HD`. In the second action, we introduce a dependency between the head and the first node with the edge label `SB`, using the edge insertion action `<root> subj #s`. Since there is only one head per nonterminal—marking more than one node as a head is a conversion error and results in immediate failure—we can refer to it using the implicit variable `<root>`. Since the conversion process is not tied to any



(a) Original (b) Conversion

Figure 1: Conversion Example

specific dependency grammar, the set of edges is not limited, and any alphanumeric string is a valid edge label.

Each node marked as head and each node on the right-hand side of an edge insertion action are marked as *consumed*. In a bound rule, each node is consumed at least once, which means that it cannot be matched in subsequent quantifiers. Consumption is also used to check the completeness of the conversion process after no more rules can be applied.

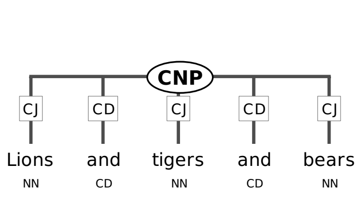
In some cases, it is necessary to drop edges that are present in the constituent tree, like punctuation tokens, which are attached to the virtual root phrase in the NEGRA annotation scheme [3]. The `ignore` action marks a node variable as consumed without introducing a dependency edge.

3.3.3 The Conversion Process

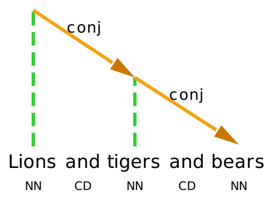
Each nonterminal p, unless all its children are dropped from the structure, must have a unique root head c_p, otherwise the conversion will fail. The head assignments of the rule applications define a surjective mapping of nonterminals to terminals, which are mapped to words in the dependency structure. As an example, we show the steps for converting the structure shown in figure 1a to the dependency graph in 1b, based on the conversion rules in example 3.

1. **Identify the heads**
   $\mathtt{NP}_1 \equiv \mathtt{t}_2 \wedge \mathtt{S}_1 \equiv \mathtt{t}_3$
2. **Create dependency edges**
   $\mathtt{t}_2 \xrightarrow{\mathtt{det}} \mathtt{t}_1 \wedge \mathtt{t}_3 \xrightarrow{\mathtt{subj}} \mathtt{NP}_1$
3. **Replace all nonterminals with an equivalent terminal**
   $\mathtt{t}_2 \xrightarrow{\mathtt{det}} \mathtt{t}_1 \wedge \mathtt{t}_3 \xrightarrow{\mathtt{subj}} \mathtt{t}_2$
4. **Replace constituent tree terminals by words**
   $\mathrm{w}_2 \xrightarrow{\mathrm{det}} \mathrm{w}_1 \wedge \mathrm{w}_3 \xrightarrow{\mathrm{subj}} \mathrm{w}_2$

In the first step, we assign a head to each nonterminal phrase, for instance “dog” (t_2) for NP_1. In step 2, all remaining nodes are connected to their heads using dependency edges. In step 3, all nonterminals in the dependency edge definitions are replaced by equivalent terminals, which are determined using the assignments



(a) Original



(b) Conversion

Figure 2: Handling of Coordination

from step 1. The last step replaces all terminals by the equivalent words in the dependency graph.

3.4 Free Rules

Free rules allow for conversion of arbitrarily large structures like coordination phrases. Consider the tree in figure 2a; instead of selecting one conjunct as the head and connecting all other conjuncts to it, it should also be possible to create a chained structure as seen in figure 2b. Since there is no limit to the number of conjuncts, we combine a bound rule and a free rule that invokes itself recursively, as shown in example 4.

(4)
```
rule coord() {
    first #first { CJ [FREC] } => root #first;
    first? #e { CJ [FREC] } => <root> conj coord();
}
context { #cnp:[cat=("CAC"|...|"CVZ")] } {
    each #cd { CD [FREC] } => ignore #cd;
    root coord();
}
```

In the bound rule, any conjunctions (edge label CD) are dropped from the phrase and the free rule `coord` is invoked unconditionally. Like bound rules, free rules have at most one node that is marked as a head, again using the `root` action, which is also used as the final argument to the action in which the free rule is invoked. Therefore, the head of the first invocation of `coord` also becomes the head of the whole coordination phrase.

In `coord`, the leftmost conjunct (edge label CJ) is marked as the head. The second action, quantified with `first?` is invoked only if another conjunct remains, which is connected to the head with the dependency `conj`. Otherwise, the base case is reached and recursion stops.

Free rules are also useful to avoid rule duplication when certain constituent phrases have a similar structure.

Result	Count	Percentage
Complete	43,299	85.79%
Partial[1]	5,902	11.69%
Failure	1,269	2.51%
Total	50,470	100.00%

Table 1: Conversion Coverage in the TIGER Treebank Release 2.1

3.5 Match Cascades

Some phrases have a varying structure. The head of an NP can be a noun, a personal or demonstrative pronoun or a proper name, which are mutually exclusive, but in some cases it is necessary to match some with higher priority than others. So far, this can only be approximated by using a series of quantifiers as shown in example 5. To solve this problem, we introduce cascaded descriptions as shown in example 6.

(5)
```
first? #h { [pos="NK"] } => root #h;
first  #h { [pos="FM"] } => root #h;
```

(6)
```
first #h { [pos="NK"], [pos="FM"] } => root #h;
```

The descriptions in example 6 are matched against the available nodes in the order they are written and behave similar to the rules in ex. 5, but only one head is assigned.

4 Conversion of the TIGER Treebank

As a first experiment, we wrote a rule set for converting the German TIGER treebank [2] into a dependency format. The rule set consists of 14 bound and 5 free rules, with little more than 100 individual actions. Coverage is shown in table 1, which is based on a version of the converter that handles secondary edges transparently. Any secondary edge in the constituent graph is treated like a dominance edge, but is eventually inserted as a secondary dependency. If deactivated, the failure rate raises to 3.51%.

We used the interactive character of our tool to constantly check and improve performance and coverage. We did not perform an exhaustive evaluation, but only errors remain.

5 Conclusion & Future Work

In this paper, we created a declarative formalism for converting constituent trees into dependency graphs and showed that the formalism is already powerful enough

[1]Conversion process results in several unconnected substructures.

to convert more than 97% of the TIGER treebank. Rule writing and debugging is supported by the graphical interface which converts on the fly and delivers error and coverage messages.

We implemented the formalism in *ladon*[2], a library for structured linguistic annotation data. We also developed a new tool for browsing and searching treebanks that supports interactive writing of conversion rule sets as well as transparent conversion and searching.

5.1 Converting the Penn Treebank

Following the work by [9] and [14], we will create a rule set for the Penn Treebank. In the current implementation, the context for bound rules is very limited, since only local features of the node can be used. In some cases, it is desirable to choose conversion rules based on a larger context, and to be able to add further sources of information. Local conversions, for example in [12] or extensions of it like in [13] can run into linguistic problems and inconsistencies, which are largely due to the Penn Treebank annotation scheme. For example, the distinction between objects, adjuncts and indirect objects is not always possible, the distinction between appositions and conjunctions needs non-local context, and the distinction between PP complements and adjuncts is partly underspecified in the Penn Treebank.

References

[1] Adriane Boyd, Markus Dickinson, and Detmar Meurers. On Representing Dependency Relations – Insights from Converting the German TiGerDB. In *Proceedings of the Sixth Workshop on Treebanks and Linguistic Theories (TLT 2007)*, pages 31–42, Bergen, Norway, 2007.

[2] Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. The TIGER Treebank. In *Proceedings of the First Workshop on Treebanks and Linguistic Theories, TLT 2002*, Sozopol, Bulgaria, 2002.

[3] Thorsten Brants, Roland Hendriks, Sabine Kramp, Brigitte Krenn, Cordula Preis, Wojciech Skut, and Hans Uszkoreit. Das NEGRA-Annotationsschema. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[4] Michael A. Covington. GB theory as Dependency Grammar. Technical Report AI1992-03, University of Georgia, Athens, Georgia, 1992.

[5] Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *COLING 2008 Workshop on Cross-framework and Cross-domain Parser Evaluation*, Manchester, UK, 2008.

[2]http://www.cl.uzh.ch/kitt/hg

[6] Martin Forst, Nria Bertomeu, Berthold Crysmann, Frederik Fouvry, Silvia Hansen-Schirra, and Valia Kordoni. Towards a dependency-based gold standard for German parsers – The TiGer Dependency Bank. In *Proceedings of LINC-04*, Geneva, Switzerland, 2004.

[7] Katri Haverinen, Filip Ginter, Sampo Pyysalo, and Tapio Salakoski. Accurate conversion of dependency parses: targeting the Stanford scheme. In *Proceedings of Third International Symposium on Semantic Mining in Biomedicine (SMBM 2008)*, Turku, Finland, 2008.

[8] Reiko Heckel. Graph Transformation in a Nutshell. In *Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT 2004) of the SegraVis Research Training Network*, volume 148, pages 187–198. Elsevier, 2006.

[9] Richard Johansson. *Dependency-based Semantic Analysis of Natural-language Text*. PhD thesis, Department of Computer Science, Lund University, Lund, Sweden, 2008.

[10] Wolfgang Lezius. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. PhD thesis, IMS, University of Stuttgart, Stuttgart, Germany, December 2002.

[11] David Magerman. Statistical decision-tree models for parsing. In *Proceedings of the 33rd Meeting of the Association for Computational Linguistics*, Boston, MA, 1995.

[12] Joakim Nivre. *Inductive Dependency Parsing*. Springer, 2006.

[13] Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, 2007.

[14] Gerold Schneider. Extracting and Using Trace-Free Functional Dependencies from the Penn Treebank to Reduce Parsing Complexity. In *Proceedings of Treebanks and Linguistic Theories (TLT)*, Vaxjö, Sweden, 2003. University Press.

[15] Gerold Schneider. *Hybrid long-distance functional dependency parsing*. PhD thesis, University of Zürich, 2008.