Dependency Schema Transformation with Tree Transducers

Felix Hennig and **Arne Köhn**
Department of Informatics
Universität Hamburg
{3hennig, koehn}@informatik.uni-hamburg.de

Abstract

The problem of (semi-)automatic treebank conversion arises when converting between different schemas, such as from a language specific schema to Universal Dependencies, or when converting from one Universal Dependencies version to the next. We propose a formalism based on top-down tree transducers to convert dependency trees. Building on a well-defined mechanism yields a robust transformation system with clear semantics for rules and which guarantees that every transformation step results in a well formed tree, in contrast to previously proposed solutions. The rules only depend on the local context of the node to convert and rely on the dependency labels as well as the PoS tags. To exemplify the efficiency of our approach, we created a rule set based on only 45 manually transformed sentences from the Hamburg Dependency Treebank. These rules can already transform annotations with both coverage and precision of more than 90%.

1 Introduction

Since the inception of the Universal Dependencies project (McDonald et al., 2013), a number of treebanks were converted from language specific schemas to UD, in an effort to create a large multi-lingual corpus. More recently, the whole UD corpus was converted to a revised annotation schema. As treebanks and their syntax annotation guidelines are not static but are evolving constantly, converting between different schemas or modifying some parts of an existing one is a recurring problem and as such needs a reliable and systematic solution. Annotating a whole treebank again manually for a different schema is neither viable due to the sheer amount of work needed, nor necessary, since the new annotation can mostly be deduced from the syntactic information in the existing schema. Therefore, usually semi-automatic conversion is used. However, most of the time only the resulting treebank is of interest and small throwaway scripts are created just for this one conversion at hand which can not be easily used for other conversions.

We introduce a conversion system which makes it easy to specify conversion rules based on the local context of a node (i. e. word) in a dependency tree. It is based on tree transducers, a well-defined formalism already used with phrase structure trees.

Because the syntactic information is only transformed into a different representation, a rule-based approach based on local information works particularly well, as reappearing structures in the source schema correspond to reappearing structures in the generated schema. By using tree transducers, it is easy to extract rules from a small initial set of manually transformed annotations which can then be applied to the whole corpus. The rule applications can be manually observed and verified, and the ruleset can be refined based on these observations.

2 Related Work

While tree transducers have been around for a while, they have not been used with dependency trees so far. Treebank conversion itself is a topic which has gained attention only recently in association with the Universal Dependencies project.

2.1 Treebank Conversion

The UD project included six languages initially (McDonald et al., 2013) and since its inception, a number of treebanks with native annotation schemas have been converted to Universal Dependencies (e.g. Danish (Johannsen et al., 2015), Norwegian (Øvrelid and Hohle, 2016), Swedish (Nivre, 2014; Ahrenberg, 2015) and Hindi (Tandon et al., 2016)). By establishing a new annotation scheme which is also still evolving, (semi-)automatic con-

version is gaining importance, yet no common system or supporting framework was established.

However, a common pattern can be observed across conversion techniques: Usually, conversion starts with the treatment of language specific oddities such as splitting tokens into multiple syntactic words followed by conversion of PoS tags which can mostly be done on a simple word for word basis, and conversion of dependency labels combined with restructuring of the tree.

For tree restructuring and dependency label conversion, rules matching and converting parts of the tree have been proposed in different approaches. The Finnish Turku Dependency Treebank was converted with dep2dep[1], a tool based on constraint rules which are converted to Prolog (Pyysalo et al., 2015). Tyers and Sheyanova (2017) convert North Sámi to UD with a rule pipeline implemented as XSLT rules. Both approaches match edges based on node features and local tree context, matching is done anywhere in the tree, in contrast to top-down conversion in a tree transducer approach.

Ribeyre et al. (2012) outline a two step process where the rules to be applied are determined in the first step and are then iteratively applied in the second with meta-rules handling conflicting rule applications. Neither of these approaches can inherently guarantee well-formed tree output.

For the transition from UD version 1 to version 2 a script has been published based on the udapi framework, which simplifies working with CoNLL-U data.[2] All transformations in the transition script are encoded directly into the source code, making adaptations difficult and error-prone.

2.2 Applications of Tree Transducers

Thatcher (1970) extended the concept of sequential transducers to trees and already mentioned its potential usefulness for the analysis of natural language. Transducers in general can often be found in natural language processing tasks, such as grapheme to phoneme conversion, automatic speech recognition (Mohri et al., 2002) or machine translation (Maletti, 2010), however a finite state transducer is limited to reading its input left-to-right, in a sequential manner.

In machine translation tree transducers can be used, where they operate on constituency trees. The tree is traversed top to bottom and thus subtrees in the sentence can be reordered to accommodate for differences in the grammatical structure of source and target language. In a similar manner, tree transducers can be used for semantic parsing (Jones et al., 2011), converting a syntax input tree to a tree structure representing the semantic information contained in the input.

A constituency tree consists of internal nodes which represent the syntactical structure of the sentence, and leaf nodes, which are the actual words of the sentence. For dependency trees however, the internal nodes as well as the leaves are already the words of the sentence and syntactical structure is represented using the edges in the tree, which carry labels. To our knowledge, tree transducers were not used with dependency trees so far. The model has to be adapted in some ways to accommodate for the differences between dependency trees and constituency trees (see Section 3.2).

3 Tree Transducers for Dependency Conversion

Tree transducers, or more specifically top-down tree transducers, are automata which convert tree structures from the root to the leaves using rules. A rule matches parts of the tree and is therefore ideal in applications where the conversion of the whole tree can be decomposed into smaller conversion steps that rely only on local context. In constituency trees as well as dependency trees, parts of a sentence are grouped together to form a single syntactic unit, such as the subject of a sentence consisting of a noun with a determiner and an adjective or even a relative clause. When converting the tree, the size and structure of the subtree attached as subject is irrelevant, just like the rest of the tree becomes irrelevant when looking at the subtree.

3.1 Formal Definition of a Tree Transducer

A tree transducer is a five-tuple $M = (Q, \Sigma, \Delta, I, R)$.[3] Tree transducers operate on trees. A tree consists of elements from the *input alphabet* Σ, which is a set of symbols. The *rank* of a symbol is the number of child elements it needs to have in the tree. A *ranked alphabet* is a tuple (Σ, rk) where rk is a mapping of symbols to their rank. For simplicity, the ranked alphabet is called Σ, like the set of symbols it contains. The set of

[1] `https://github.com/TurkuNLP/dep2dep`

[2] `https://github.com/udapi/`; the Java implementation is also used in our tree transducer implementation.

[3] Our notation follows (Maletti, 2010), which also offers an expanded discussion on the formal aspects of top-down tree transducers.

possible trees that can be created from the alphabet is denoted by T_Σ.

In addition to the input alphabet, a transducer also has an *output alphabet* Δ with $\Sigma \cap \Delta = \emptyset$. The translation frontier is marked using *state nodes*, which are part of the set of state nodes Q, Q is disjoint with both Σ and Δ. The state nodes are also ranked symbols. Initially, the tree to be converted consists only of input symbols. A state node from the set of *initial states* $I \subseteq Q$ is added at the root, to mark the position in the tree that needs to be converted next. With each step the frontier of state nodes is pushed further down the tree. The state nodes separate the output symbols from the input symbols which still need to be converted (see Figure 1 for an example of a tree being converted top down with multiple local conversions).

The local conversion steps are performed using rules. Each top-down tree transducer has a *rule-set* $R \subseteq Q(T_\Sigma(X)) \times T_\Delta(Q(X))$, i. e. a rule is a tuple containing two tree structures: one to match parts of the tree to be converted, one to replace the matched parts. On the left-hand side, the root of each tree is a state node. The other nodes are from the input alphabet Σ. Leaf nodes may also be from the set of *variables* X. The state node is used as the anchor point to determine if a rule matches the current tree structure. The subtree of input nodes below the state node given on the left side of the rule needs to be identical to the subtree of input nodes in the tree to be converted. While the subtree of an extended tree transducer can be of arbitrary depth, a basic tree transducer can only contain subtrees of depth one. The variables on the left-hand side of the rule can match any node in the tree if they are in the correct relation to the rest of the tree. Subtrees below the one to be converted in this rule are matched to these variables, marking the end of the local context the rule seeks to convert.

If the left hand-side of the rule matches, the rule can be applied to the tree and the right-hand side of the rule is used to replace the matched subtree in the tree to be converted. The right-hand side of the rule consists of a tree of output symbols T_Δ. Again, leaves may be variables. Each variable needs to have a state node from the set Q as parent. Here, the same variables which were used on the left-hand side of the rule can be used again, to attach the unconverted subtrees below the new converted subtree. The newly introduced state nodes mark the subtrees described by the variables for further conversion. In this way, the frontier of state nodes is pushed down the tree.

A rule is linear and nondeleting if the variables $x_i \in X$ used on the left-hand side of the rule are neither duplicated nor deleted on the right-hand side of the rule and each variable is used only once on the left-hand side. If all the rules in R are linear and nondeleting, the transducer is linear and nondeleting.[4] For machine translations tasks, these properties ensure that no part of the sentence is deleted or duplicated.

3.2 Adaptations for Dependency Conversion

When defining an extended top-down tree transducer for dependency trees, we treat the labels of the edges as properties of the dependent, as tree transducers have no notion of labeled edges. We then use the set of input and output dependency labels for Σ and Δ, respectively. In contrast to the previously discussed formalism, the vertices in the dependency tree are not simply dependency labels, but also contain the word and its index in the sentence, among other information that needs to remain unchanged. We therefore define the input alphabet Σ not just as the set of input labels L_{in} but rather as $\Sigma \subseteq L_{in} \times \mathbb{N}$, which means that each node is a tuple of the dependency relation and a natural number serving as a node identifier (i. e. the index of the word in the current sentence). We will write n^σ as a shorthand for $(\sigma, n) \in \Sigma$.

The need for this becomes apparent when looking at the example in Figure 2. In the second conversion step, the tree is converted with a rule like this: $q(PP(PN(x_1))) \rightarrow obl(case(), q(x_1))$ which we will use as a running example throughout this section. The new structure of the dependency relations is clearly defined, but which node should receive which label cannot be inferred. The node previously attached with the *PP* relation could either receive the label *obl* and not change its position in the tree, or it could be attached below the node with the *obl* label with a *case* relation. This is why the nodes need to be identified, which is done via their respective index. The rules do not contain explicit indices, only the correspondence between the left and right-hand side of the rule is relevant. The concrete indices are substituted during the rule matching. In the following rule, n_1 and n_2 stand for abstract indices that are to be replaced with a concrete

[4] A more elaborate definition of these properties can be found in (Maletti, 2010)

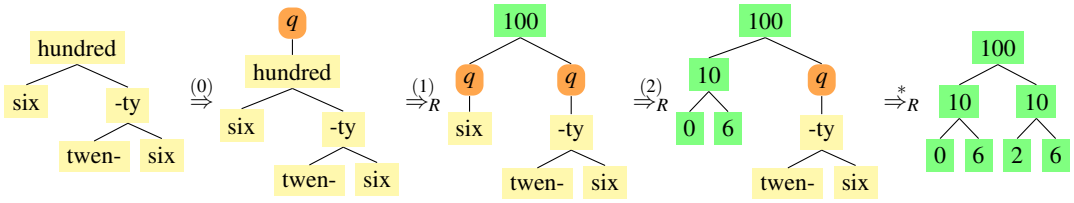

Figure 1: An example of a tree transformation of the numeric expression "six hundred twenty six" to its digit form 626, using a top-down tree transducer. The yellow nodes are nodes from the input alphabet, the green nodes are from the output alphabet and the orange nodes are state nodes. Example based on Maletti (2010), the last step includes multiple transformation steps.

index: $q(n_1^{PP}(n_2^{PN}(x_1))) \rightarrow n_2^{obl}(n_1^{case}(), q(x_1))$ It can be seen that n_1 and n_2 switch their position in the tree, so the node previously attached as *PP* got moved below the *obl* node and received the *case* relation.

After defining $\Sigma \subseteq L_{in} \times \mathbb{N}$ and $\Delta \subseteq L_{out} \times \mathbb{N}$ as described above, we can define a new property of transducers. A rule $r = (T_l, T_r)$ is *word-preserving* if it meets the following conditions: First, r is linear and nondeleting. Second, the left-hand side of r cannot contain the same node index twice: $n \neq m \; \forall (\cdot, n), (\cdot, m) \in T_l, (\cdot, n), (\cdot, m) \in \Sigma$. Third, r neither deletes nor duplicates a matched word. This means that for $N_{in} := \{n \in \mathbb{N} | (\cdot, n) \in T_l\}$ each $n \in N_{in}$ appears exactly once in T_r. A ruleset is word-preserving if all its rules are word-preserving. This property ensures that word attachments and dependency relations of the sentence can change but no word is removed or duplicated.

The input symbols of a tree transducer are ranked. A node in a dependency tree can have any number of children, therefore the alphabet in the tree can be assumed to contain each symbol with multiple different ranks to adhere to the formalism. To accommodate for the varying number of dependents in the rules, without requiring a rule for each possible number of dependents, the variable mechanism to match subtrees is extended to match multiple subtrees into one *catch-all* variable. As an example, the dependency tree in Figure 2 could have additional dependents below the "Transducer" node, such as adjectives or a relative clause. Our running example rule used in step two of the conversion process does not account for additional dependents. To account for them, the rule can be adapted as follows: $q(n_1^{PP}(n_2^{PN}(xs_1))) \rightarrow n_2^{obl}(n_1^{case}(), q(xs_1))$. Here, xs_1 can match a variable amount of nodes, hence formally it could be represented by $x_1, x_2, ..., x_n \in X$. In this way, one rule represents multiple rules with symbols of different rank, which means that the transducer can still be seen as ranked, just with a compact representation. Also in contrast to the conventional formalism, the child elements in the tree structures of the rules are not ordered and can therefore be matched to the annotation nodes in any order.

We only use one type of state node. Information about the already converted part of the tree can be accessed by matching nodes above the frontier. This way, different rules can be executed based on, for example, the dependency relation of the parent node, without the requirement to pass this information down encoded in a state node. Only direct ancestors can be matched, as these are guaranteed to have been translated already. Matching dependents of parents would introduce ordering effects based on the order in which the children of a node are converted. Because the information in the parent nodes could be encoded in a state node, accessing it by matching the parent is not a violation of the transducer formalism.

3.3 Look-ahead Extension

Similarly to this look-back mechanism, a look-ahead mechanism is required to inspect context below the current node without converting it immediately. Figure 3 contains an example for this. A *NEB* dependent, a subordinate clause, is converted differently based on its context. The subordinate clause can either be a core dependent as a *ccomp* (rule 2), or it can have an adverbial function and would be attached as *advcl* (rule 1). In this case a simple heuristic based on the presence of an object is used to distinguish core and non-core subordinate clauses. *KONJ* and *OBJA* in the first two rules are matched to constrain rule application by requiring a certain context, but the conversion of these nodes is left to be treated in additional rules. The *KONJ* dependent, a subordinate conjunction, is transformed in an additional rule (rule 3) to clearly separate the

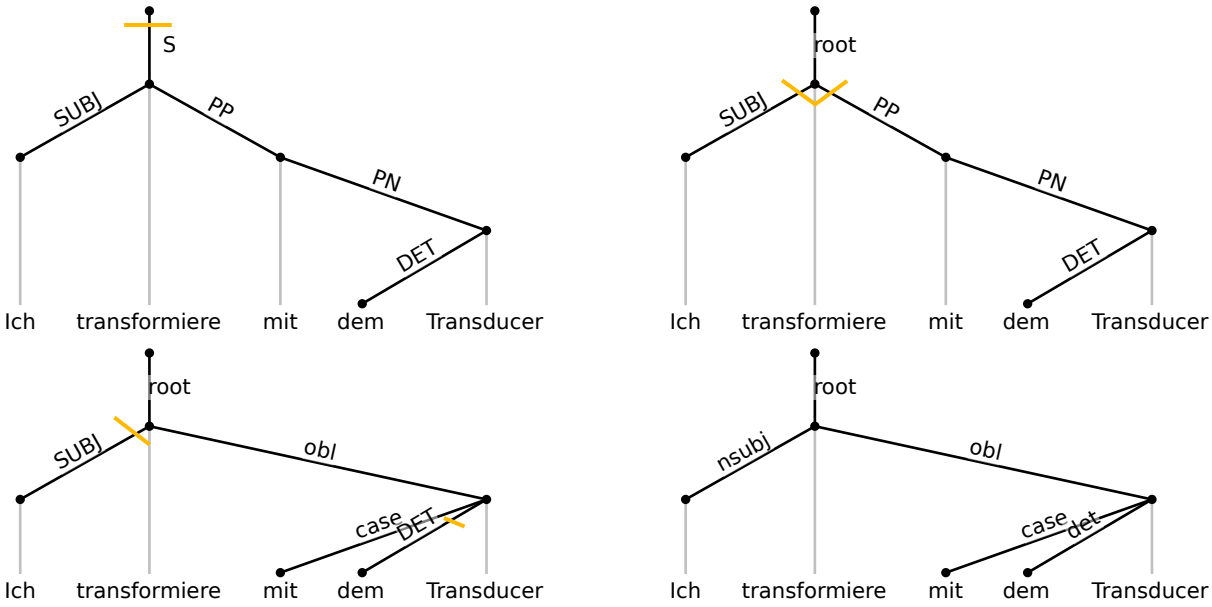

Figure 2: Conversion of "Ich transformiere mit dem Transducer" (*i transform with the transducer*); yellow lines indicate the state nodes, HDT labels are uppercase, UD are lowercase.

$$n_0(q(n_1^{NEB}(n_2^{KONJ}()), n_3^{OBJA})) \rightarrow n_0(n_1^{advcl}(q(n_2())), q(n_3())) \quad (1)$$

$$n_0(q(n_1^{NEB}(n_2^{KONJ}()))) \rightarrow n_0(n_1^{ccomp}(q(n_2()))) \quad (2)$$

$$q(n_1^{KONJ}()) \rightarrow n_1^{mark}() \quad (3)$$

Figure 3: Formal representation of tree transducer rules for transforming subordinate clauses.

conversion of different labels into different rules and to avoid duplicating conversion logic.

The *OBJA* dependent is attached with different labels based on its context, just like the *NEB* dependent in this case. Converting *NEB* and *OBJA* together in the same rule would result in a combinatorial explosion of rules, as not only would multiple rules be required to convert the *NEB* to different output symbols, but also multiple variants of the *NEB* rules for the different conversions of the *OBJA* dependent. While this separation of concerns into different rules results in a more structured and systematic ruleset, it requires rules to be tested in a specific order, as the *NEB* conversion now relies on the *OBJA* to be converted afterwards.

3.4 Cross-frontier Modifications

When converting a function-head treebank to a content-head scheme like Univeral Dependencies, edges are inverted a lot and in some cases the function and content word appear in the same pattern all the time, like the *case nmod* combination, but sometimes the structure varies more. Verb modality and tense is often expressed with auxiliary verbs or modality verbs in addition to the content bearing verb. In a function-head treebank this means there can be long chains of auxiliary verbs with the actual main verb at the bottom of the chain. The varying depths would require multiple rules for different depths of the chain, as all the function words and the content word at the end need to be converted at once, to allow for the inversion of the head. For these cases it is practical to make modifications to already converted parts of the tree.

An example of such a use case is shown in Figure 4. The *S* and *SUBJ* node are converted just like they would be if there were no auxiliary verbs. When the first auxiliary relation is converted, the already transformed root node is matched above the frontier and repositioned below the node previously attached as *AUX*, thereby inverting the relation between head and auxiliary verb. For the next auxiliary relation, the same rule is used. On the left-hand side of the rule the dependency relation of the parent is assigned to a variable $\$x$, to reassign this relation to the previous auxiliary verb. Using a

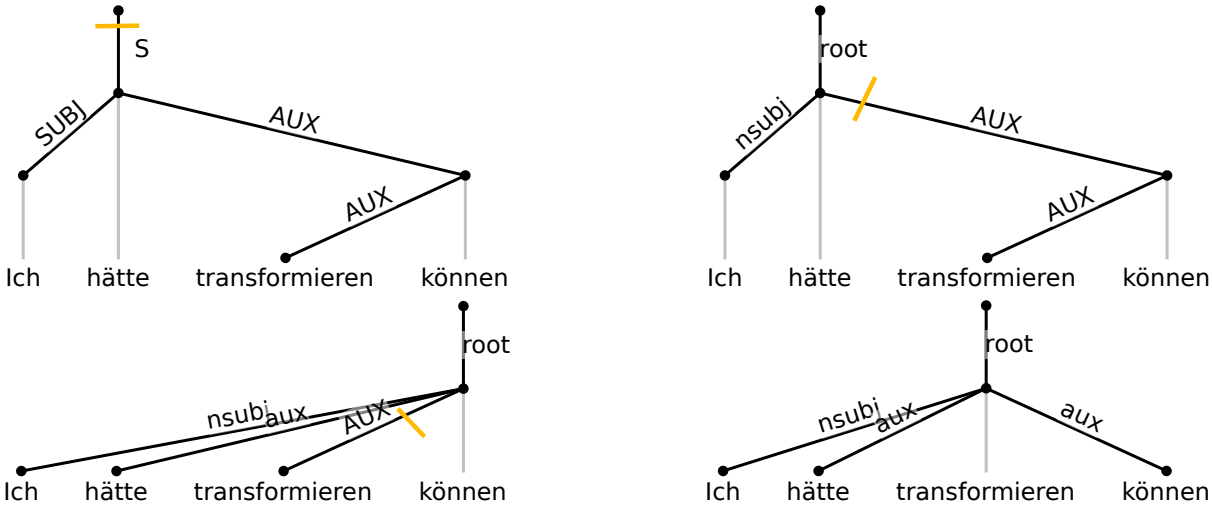

Figure 4: Conversion of a dependency tree from a function-head to a content-head scheme. The *AUX* edges are inverted when they are converted. The rule used in step 2 and 3 is $p^{\$x}(q(n^{AUX}(xs_n), xs_q), xs_p) \rightarrow n^{\$x}(p^{aux}(), q(xs_n, xs_q), xs_p)$

variable eliminates the need to have an additional rule for each possible parent relation.

The rule also contains catch-all variables below each individual node in the tree to cover all possible dependent attachments. Specifying a transformation strategy for each possible dependent makes this rule adaptable to different structures. While the first application of the rule matches "ich" in the xs_p variable and "transformieren" in the xs_n variable, the second application matches both "ich" and "hätte" in xs_p.

4 Implementation

The implementation of the transducer incorporates all properties discussed in Section 3. A transducer is specified through a single file containing the rules. The input and output alphabet are specified implicitly, by the labels used in these rules.

Figure 5 demonstrates the syntax, which is similar to the rule notation introduced in Section 3. A node is specified by an identifier and optionally its dependency relation, connected with a colon. Catch-all variables consist of an identifier prefixed by a question mark. A state node is denoted by curly braces, it does not have an identifier as there is only a single type of state node. A basic label translation is shown in (a), (b) shows the rule example discussed throughout Section 3.2 and (c) shows the rule mentioned in Figure 4.

Not all left-hand sides of the rules contain state nodes, and the right-hand sides never contain them. If a rule does not contain a state node on the left-hand side, it is assumed to be above the root node. As state nodes on the right-hand side of rules are always above non-converted nodes, these state nodes are inferred as well.

(a)
```
n:SUBJ() -> n:nsubj();
```

(b)
```
n1:PP(n2:PN(?r2), ?r1)
  -> n2:nmod(n1:case(), ?r1, ?r2);
```

(c)
```
parent:$x({n:AUX(?auxr), ?fr}, ?r)
  -> n:$x(parent:aux(), ?auxr, ?r, ?fr);
```

(d)
```
p({n:SUBJ(?r), ?fr}, ?pr)
  -> p(n:nsubj({?r}), {?fr}, ?pr)
```

(e)
```
p({n1:OBJA(), n2:OBJD()})
  -> p(n1:obj(), n2:iobj());
  n:OBJA() -> n:obj();
  n:OBJD() -> n:obj();
```

(f)
```
p.NE({n.NN:APP()}) -> p(n:appos());
p.NN({n.NN:APP()}) -> p(n:compound());
```

(g)
```
p({n:APP()}) -> p(n:compound()) :-
  {n.getOrd() < p.getOrd()};
```

Figure 5: Rule (a) to (c) show rule syntax examples and (d) is a verbose version of (a). Rule (e) and (f) show rule combinations and (g) shows the use of groovy code to further constrain rule applicability.

The `n` node in (a) does not have a variable node as a dependent like the `n1` and `n2` node in (b). Whenever a node does not have a catch-all dependent, it is assumed that potential dependents should remain attached the way they are, not that the node should not have dependents at all. Therefore, it

is only necessary to use catch-all variables if the words matched by the variable should be reattached somewhere else. For example, the `?r1` catch-all variable in (b) is necessary, whereas `?r2` could be omitted – it is only stated explicitly to avoid confusion about the location of the `n2` dependents. Rule (a) as used in the transducer after inference of all additional parts mentioned is shown in (d): Above the `n` node, a frontier and a parent node are inferred. In addition, the `n` node as well as the frontier and parent node each receive a catch-all variable.

The rules are tested in the order in which they are written in the file. Each rule is tested at each frontier node, and if it cannot be applied anywhere, the next rule is tested. Rules which only apply within a narrow context and describe exceptions to a general rule appear above these generic fall-back rules. This is exemplified in (e), where the first rule covers the specific case of ditransitive verbs, attaching one object as `obj` and the other as `iobj`, based on their grammatical case. The other two rules cover the common case of simple transitive verbs with a single object only.

The Part-of-Speech tags are important to distinguish structures with otherwise identical dependency relations. They can be accessed directly via a period after the node identifier, as illustrated in (f). The PoS tags are only used to constrain rule applicability and cannot be set on the right-hand side of a rule.

Lastly, arbitrary groovy[5] code can added to the rules to further constrain matching or even to modify the resulting tree. Rule (g) shows an example where linear order in the sentence is checked using groovy. In the groovy code the tree before and after the transformation can be accessed and modified, allowing to formulate additional constraints or modify the resulting tree, to for example add feats to the nodes.

5 Experiments and Results

To evaluate the feasibility of the tree transducer approach to treebank conversion, a ruleset for the conversion of the Hamburg Dependency Treebank (Foth et al., 2014) to Universal Dependencies was created and applied to the treebank. Due to its size of more than 200k manually annotated sentences, a conversion needs to be streamlined as much as possible.

[5]Groovy Language: `http://groovy-lang.org/`

5.1 Ruleset

To get familiar with both the HDT and UD tagset, a sample subset of 45 sentences was chosen based on the requirement that each dependency relation from the HDT tagset appears at least three times in the selected dependency trees. Each sentence was converted manually to UD and notes of reappearing patterns or difficult and unusual relation structures were taken. The notes and knowledge of both schemas was then used to create an initial ruleset, which was tested on the previously annotated trees. In an iterative manner the ruleset was refined by comparing the generated results against the previously manually converted trees.

It took about a week of work to annotate the trees, take notes and create the ruleset. This was largely due to being unfamiliar with the HDT as well as the UD tagset and not knowing which features the software should even have. The software was also adapted in this time to incorporate new features which were deemed necessary while the rules were created. The resulting ruleset contains 58 rules, each with a complexity similar to the ones shown in Figure 5. Having the conversion software already, as well as previous knowledge about the source and target annotation schema, a ruleset with similar effectiveness can be created in one or two days.

As the HDT annotation guidelines do not include punctuation, it is always attached to the root and no information about the potential attachment in the tree using the UD schema can be inferred from the local context. Therefore, punctuation was ignored in the experiment.

5.2 Evaluation

We used the ruleset to convert part B of the treebank, containing about 100k sentences. 91.5% of the words were converted successfully.

To evaluate the correctness of the conversions, we used 50 sentences manually converted to UD. These sentences were chosen randomly from the treebank, excluding the sentences used to create the ruleset. The sentence annotations were converted with the ruleset and compared to the manually annotated ones. Out of 698 words in total, 36 words were not converted and 51 words were converted incorrectly, yielding a precision of 92% and a recall of 94%.

The transducer also uncovered a few annotation errors in the manually annotated trees. For example, `nmod` relations on words which should have

been attached as obl. These errors in the target data were corrected before calculating the values mentioned above, as they would distort the actual evaluation results of the transducer.

6 Analysis

Converting a node requires that the node and its context can be matched by a rule in the ruleset. This means that adding rules with a narrow context – and therefore wide applicability – as fall-back rules to more specific transformations will increase the coverage of the ruleset, an effect that is amplified by the fact that the descendants of an unconvertible node cannot be converted as well, as the frontier cannot move beyond a node for which no conversion rule exists. About half of the unconverted nodes were descendants of an unconvertible node and not necessarily unconvertible themselves. The ruleset contains rules which make strict assumptions about the PoS tags of the nodes, specifically in the rules concerning the conversion of appositions, as well as assuming that certain dependency relations always appear together, such as a conjunct and a coordinating conjunction. In both cases, adding fall-back rules with less context will increase coverage.

On the other hand, to increase precision, some rules need to be more constrained by including more context. This is especially the case where distinctions need to be made in the target schema which are not encoded in the source schema. For example, the UD schema makes a distinction between appos, compound, and flat. These syntactic relations are all grouped under the APP label in the HDT annotation schema. Also, the distinction between obl and nmod as well as between advcl and ccomp has no correspondence in the HDT schema. While these distinctions are difficult to make, it is most of the time possible to distinguish between the cases by looking at the PoS tags of the nodes and their parents or inspecting the dependents through look-ahead.

Some conversions are exceptions to the rule and cannot be converted automatically. This is the case with multiword expressions, a concept not used in the HDT. Different multiword expressions consist of different types of words, each German multiword expression would require a specific rule based on the word forms. Also, reflexive pronouns attached to inherently reflexive verbs should not be attached as objects in UD, but as expl. This is not the case in the HDT, and as it cannot be inferred from the structural context if a verb is inherently reflexive or not, it is impossible to convert these relations correctly in an automated transformation process. These cases can be decided by a human annotator, prompting to the implementation of an interactive conversion system (see Section 7).

Large parts of the conversion worked very well, such as the conversion of different types of objects to obj and iobj, distinguishing nmod and obl, and amod and nummod. Distinguishing advcl and ccomp worked in most cases, but some cases are also hard to decide for a human. The inversion of function and content head worked well, such as switching case and nmod nodes or inverting aux relations. A relevant portion of the HDT labels had a correct correspondence to a UD label and could therefore be converted easily.

7 Conclusion and Outlook

We introduced a framework which makes it possible to write a useful tree transducer for dependency schema transformation based on a very small amount of manually transformed annotations in little time. An approach relying on converting groups of words in a local context fits the structure of natural language, where functional units in a sentence often consist of multiple sub groups of words. By relying on the tree transducer framework, the rule writer can focus on the conversion itself and does not need to worry about termination or preserving the tree structure. In addition, no programming skills are needed for writing transformation rules.

The experiments performed indicate the need of an interactive transformation mode: While detecting ambiguous structures is possible, deciding them automatically is hard. As such, semi-interactive conversion is the next step, showing the intermediate conversion results of rules that are not fully reliable to a human annotator and allowing her to chose whether to perform this step or even to restructure the tree manually before continuing with automatic conversion.

For the conversion of the HDT into UD, more time needs to be invested to refine the rules, and the final conversion should be done interactively.

Code and data is available under:
http://nats.gitlab.io/truducer

Acknowledgements We would like to thank the anonymous reviewers for helpful comments.

References

Lars Ahrenberg. 2015. Converting an english-swedish parallel treebank to universal dependencies. In *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)*, pages 10–19, Uppsala, Sweden, August. Uppsala University, Uppsala, Sweden.

Kilian A. Foth, Arne Köhn, Niels Beuck, and Wolfgang Menzel. 2014. Because size does matter: The Hamburg Dependency Treebank. In *Proceedings of the Language Resources and Evaluation Conference 2014*. LREC, European Language Resources Association (ELRA).

Anders Johannsen, Héctor Martínez Alonso, and Barbara Plank. 2015. Universal dependencies for danish. In Markus Dickinson, Erhard Hinrichs, Agnieszka Patejuk, and Adam Przepiórkowski, editors, *Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT14)*, pages 157–167, Warsaw, Poland.

Bevan Jones, Mark Johnson, and Sharon Goldwater. 2011. Formalizing semantic parsing with tree transducers. In *Proceedings of the Australasian Language Technology Association Workshop 2011*, pages 19–28, Canberra, Australia, December.

Andreas Maletti. 2010. Survey: Tree transducers in machine translation. In Henning Bordihn, Rudolf Freund, Thomas Hinze, Markus Holzer, Martin Kutrib, and Friedrich Otto, editors, *Proc. 2nd Int. Workshop Non-Classical Models of Automata and Applications*, volume 263 of `books@ocg.at`, pages 11–32. Österreichische Computer Gesellschaft.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria, August. Association for Computational Linguistics.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69 – 88.

Joakim Nivre. 2014. Universal dependencies for swedish. In *Proceedings of the Swedish Language Technology Conference (SLTC)*, Uppsala, Sweden, November. Uppsala University, Uppsala, Sweden.

Lilja Øvrelid and Petter Hohle. 2016. Universal dependencies for norwegian. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Sara Goggi, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asunción Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016*. European Language Resources Association (ELRA).

Sampo Pyysalo, Jenna Kanerva, Anna Missilä, Veronika Laippala, and Filip Ginter. 2015. Universal dependencies for finnish. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*, pages 163–172, Vilnius, Lithuania, May. Linköping University Electronic Press, Sweden.

Corentin Ribeyre, Djamé Seddah, and Éric Villemonte de la Clergerie. 2012. A linguistically-motivated 2-stage tree to graph transformation. In *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11)*, pages 214–222, Paris, France, September.

Juhi Tandon, Himani Chaudhry, Riyaz Ahmad Bhat, and Dipti Misra Sharma. 2016. Conversion from paninian karakas to universal dependencies for hindi dependency treebank. In Katrin Tomanek and Annemarie Friedrich, editors, *Proceedings of the 10th Linguistic Annotation Workshop held in conjunction with ACL 2016, LAW@ACL 2016, August 11, 2016, Berlin, Germany*. The Association for Computer Linguistics.

James W. Thatcher. 1970. Generalized sequential machine maps. *Journal of Computer and System Sciences*, 4(4):339–367.

Francis M. Tyers and Mariya Sheyanova. 2017. Annotation schemes in north sámi dependency parsing. In *Proceedings of the Third Workshop on Computational Linguistics for Uralic Languages*, pages 66–75, St. Petersburg, Russia, January. Association for Computational Linguistics.