From Universal Dependencies to Abstract Syntax

Aarne Ranta
University of Gothenburg
aarne@chalmers.se

Prasanth Kolachina
University of Gothenburg
prasanth.kolachina@gu.se

Abstract

Abstract syntax is a tectogrammatical tree representation, which can be shared between languages. It is used for programming languages in compilers, and has been adapted to natural languages in GF (Grammatical Framework). Recent work has shown how GF trees can be converted to UD trees, making it possible to generate parallel synthetic treebanks for those 30 languages that are currently covered by GF. This paper attempts to invert the mapping: take UD trees from standard treebanks and reconstruct GF trees from them. Such a conversion is potentially useful in bootstrapping treebanks by translation. It can also help GF-based interlingual translation by providing a robust, efficient front end. However, since UD trees are based on natural (as opposed to generated) data and built manually or by machine learning (as opposed to rules), the conversion is not trivial. This paper will present a basic algorithm, which is essentially based on inverting the GF to UD conversion. This method enables covering around 70% of nodes, and the rest can be covered by approximative back up strategies. Analysing the reasons of the incompleteness reveals structures missing in GF grammars, but also some problems in UD treebanks.

1 Introduction

GF (Grammatical Framework (Ranta, 2011)) is a formalism for multilingual grammars. Similarly to UD (Universal Dependencies, (Nivre et al., 2016)), GF uses shared syntactic descriptions for multiple languages. In GF, this is achieved by using **abstract syntax trees**, similar to the internal representations used in compilers and to Curry's tectogrammatical formulas (Curry, 1961). Given an abstract syntax tree, strings in different languages can be derived mechanically by **linearization functions** written for that language, similar to pretty-printing rules in compilers and to Curry's phenogrammatical rules. The linearization functions of GF are by design reversible to parsers, which convert strings to abstract syntax trees. Figure 1 gives a very brief summary of GF to readers unfamiliar with GF.

In UD, the shared descriptions are dependency labels and part of speech tags used in dependency trees. The words in the leaves of UD trees are language-specific, and languages can extend the core tagset and labels to annotate constructions in the language. The relation between trees and strings is not defined by grammar rules, but by constructing a set of example trees—a treebank. From a treebank, a parser is typically constructed by machine learning (Nivre, 2006). There is no mechanical way to translate a UD tree from one language to other languages. But such a translation can be approximated in different ways to bootstrap treebanks (Tiedemann and Agic, 2016).

GF's linearization can convert abstract syntax trees to UD trees (Kolachina and Ranta, 2016). This conversion can be used for generating multilingual (and parallel) treebanks from a given set of GF trees. However, to reach the full potential of the GF-UD correspondence, it would also be useful to go to the opposite direction, to convert UD trees to GF trees. Then one could translate standard UD treebanks to new languages. One could also use dependency parsing as a robust front-end to a translator, which uses GF linearization as a grammaticality-preserving backend (Angelov et al., 2014), or to a logical form generator in the style of (Reddy et al., 2016), but where GF trees give an accurate intermediate representation in the style of (Ranta, 2004). Figure 2 shows both of these scenarios, using the term **gf2ud** for the con-

The abstract syntax defines a set of **categories**, such as CN (Common Noun) and AP (Adjectival Phrase), and a set of **functions**, such as ModCN (modification of CN with AP):

```
cat CN ; AP
fun ModCN : AP -> CN -> CN
```

A concrete syntax defines, for each category, a **linearization type**, and for each function, a **linearization function**; these can make use of **parameters**. For English, we need a parameter type Number (singular or plural). We define CN as a **table** (similar to an inflection table), which produces a string as a function Number (`Number=>Str`). As AP is not inflected, it is just a string. Adjectival modification places the AP before the CN, passing the number to the CN head of the construction:

```
param Number = Sg | Pl
lincat CN = Number => Str
lincat AP = Str
lin ModCN ap cn = \\n => ap ++ cn ! n
```

In French, we also need the parameter of gender. An AP depends on both gender and number. A CN has a table on Number like in English, but in addition, an inherent gender. The table and the gender are collected into a **record**. Adjectival modification places the AP after the CN, passing the inherent gender of the CN head to the AP, and the number to both constituents:

```
param Gender = Masc | Fem
lincat CN = {s : Number => Str ; g : Gender}
lincat AP = Gender => Number => Str
lin ModCN ap cn = {
  s = \\n => cn ! n ++ ap ! cn.g ! n ;
  g = cn.g
  }
```

Context-free grammars correspond to a special case of GF where Str is the only linearization type. The use of tables (`P=>T`) and records (`{a : A ; b : B}`) makes GF more expressive than context-free grammars. The distinction between dependent and inherent features, as well as the restriction of tables to finite parameter types, makes GF less expressive than unification grammars. Formally, GF is equivalent to PMCFG (Parallel Multiple Context-Free Grammars) (Seki et al., 1991), as shown in (Ljunglöf, 2004), and has polynomial parsing complexity. The power of PMCFG has shown to be what is needed to share an abstract syntax across languages. In addition to morphological variation and agreement, it permits discontinuous constituents (used heavily e.g. in German) and reduplication (used e.g. in Chinese questions). The GF Resource Grammar Library uses a shared abstract syntax for currently 32 languages (Indo-European, Fenno-Ugric, Semitic and East Asian) written by over 50 contributors.
Software, grammars, and documentation are available in `http://www.grammaticalframework.org`

Figure 1: GF in a nutshell. The text works out a simple GF grammar of adjectival modification in English and French, showing how the structure can be shared despite differences in word order and agreement.

version of Kolachina and Ranta (2016) and **ud2gf** for the inverse procedure, which is the topic of this paper.

GF was originally designed for multilingual generation in controlled language scenarios, not for wide-coverage parsing. The GF Resource Grammar Library (Ranta, 2009) thus does not cover everything in all languages, but just a "semantically complete subset", in the sense that it provides ways to express all kinds of content, but not necessarily all possible ways to express it. It is has therefore been interesting to see how much of the syntax in UD treebanks is actually covered, to assess the completeness of the library. In the other direction, some of the difficulties in ud2gf mapping suggest that UD does not always annotate syntax in the most logical way, or in a way that is maximally general across languages.

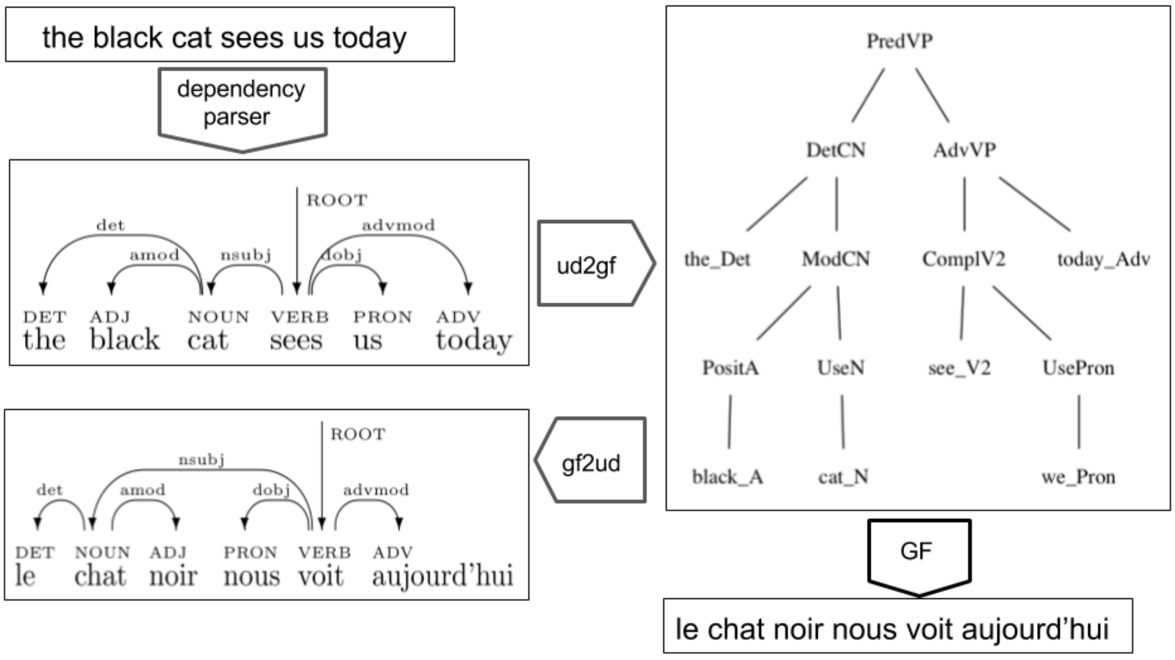
The work reported in this paper is the current status of work in progress. Therefore the results are not conclusive: in particular, we expect to improve the missing coverage in a straightforward way. The most stable part of the work is the annotation algorithm described in Sections 3 an 4. It is based on a general notation for dependency configurations, which can be applied to any GF grammar and to any dependency annotation scheme—not only to the UD scheme. The code for the algorithm and the annotations used in experiments is available open source.[1]

The structure of the paper is as follows: Section 2 summarizes the existing gf2ud conversion and formulates the problem of inverting it. Section 3 describes a baseline bottom-up algorithm for translation from UD trees to GF trees. Section 4 presents some refinements to the basic algorithm. Section 5 shows a preliminary evaluation with UD treebanks for English, Finnish, and Swedish. Section 6 concludes.

2 From gf2ud to ud2gf

The relation between UD and GF is defined declaratively by a set of **dependency configurations**. These configurations specify the dependency labels that attach to each subtree in a GF tree. Figure 3 shows an abstract syntax specification to-

[1] `https://github.com/GrammaticalFramework/gf-contrib/tree/master/ud2gf`



the black cat sees us today
dependency parser
ROOT
det
amod
nsubj
advmod
dobj
DET ADJ NOUN VERB PRON ADV
the black cat sees us today
ud2gf
gf2ud
ROOT
nsubj
det
amod
dobj
advmod
DET NOUN ADJ PRON VERB ADV
le chat noir nous voit aujourd'hui
PredVP
DetCN
AdvVP
the_Det
ModCN
ComplV2
today_Adv
PositA
UseN
see_V2
UsePron
black_A
cat_N
we_Pron
GF
le chat noir nous voit aujourd'hui

Figure 2: Conversions between UD trees, GF trees, and surface strings in English and French.

gether with a dependency configuration, as well as a GF tree with corresponding labels attached.

Consider, for example, the second line of the "abstract syntax" part of Figure 3, with the symbol `ComplV2`. This symbol is one of the **functions** that are used for building the abstract syntax tree. Such a function takes a number of trees (zero or more) as arguments and combines them to a larger tree. Thus `ComplV2` takes a `V2` tree (two-place verb) and an `NP` tree (noun phrase) to construct a `VP` tree (verb phrase). Its name hints that it performs complementation, i.e. combines verbs with their complements. Its dependency configuration `head dobj` specifies that the first argument (the verb) will contain the label `head` in UD, whereas the second argument (the noun phrase) will contain the label `dobj` (direct object). When the configuration is applied to a tree, the `head` labels are omitted, since they are the default. Notice that the order of arguments in an abstract syntax tree is independent of the order of words in its linearizations. Thus, in Figure 2, the object is placed after the verb in English but before the verb in French.

The algorithm for deriving the UD tree from the annotated GF tree is simple:

- for each leaf *X* (which corresponds to a lexical item)
  - follow the path up towards the root until you encounter a label *L*
  - from the node immediately above *L*, follow the **spine** (the unlabelled branches) down to another leaf *Y*
  - *Y* is the head of *X* with label *L*

It is easy to verify that the UD trees in Figure 2 can be obtained in this way, together with the English and French linearization rules that produce the surface words and the word order. In addition to the configurations of functions, we need **category configurations**, which map GF types to UD part of speech (POS) tags.

This algorithm covers what Kolachina and Ranta (2016) call **local abstract configurations**. They are sufficient for most cases of the gf2ud conversion, and have the virtue of being compositional and exactly the same for all languages. However, since the syntactic analysis of GF and UD are not exactly the same, and within UD can moreover differ between languages, some **non-local** and **concrete** configurations are needed in addition. We will return to these after showing how the local abstract configurations are used in ud2gf.

The path from GF trees to UD trees (**gf2ud**) is deterministic: it is just linearization to an annotated string representing a dependency tree. It defines a relation between GF trees and UD trees: *GF tree t produces UD tree u*. Since the mapping involves loss of information, it is many-to-

abstract syntax	dependency configuration
`PredVP  : NP   -> VP  -> Cl`	`nsubj head`
`ComplV2 : V2   -> NP  -> VP`	`head dobj`
`AdvVP   : VP   -> Adv -> VP`	`head advmod`
`DetCN   : Det  -> CN  -> NP`	`det head`
`ModCN   : AP   -> CN  -> CN`	`amod head`
`UseN    : N    -> CN`	`head`
`UsePron : Pron -> NP`	`head`
`PositA  : A    -> AP`	`Head`
cat A	ADJ
cat Adv	ADV
cat Det	DET
cat N	NOUN
cat Pron	PRON
cat V2	VERB

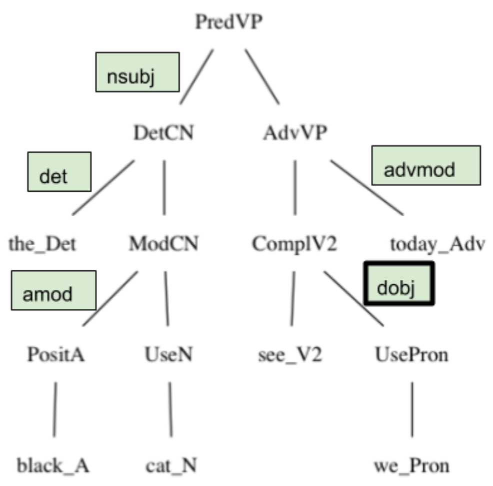


Figure 3: Annotating a GF tree with dependency labels. The label `dobj` results from the annotation of the `ComplV2` function. The category annotation (`cat`) are used in Figures 1 and 3 to map between GF categories and UD POS tags.

one. The opposite direction, ud2gf, is a nondeterministic search problem: *given a UD tree u, find all GF trees t that can produce u.* The first problem we have to solve is thus

> **Ambiguity**: a UD tree can correspond to many GF trees.

More problems are caused by the fact that GF trees are formally generated by a grammar whereas UD trees have no grammar. Thus a UD tree may lack a corresponding GF tree for many different reasons:

> **Incompleteness**: the GF grammar is incomplete.
> **Noise**: the UD tree has annotation errors.
> **Ungrammaticality**: the original sentence has grammar errors.

Coping with these problems requires **robustness** of the ud2gf conversion. The situation is similar to the problems encountered when GF is used for wide-coverage parsing and translation (Angelov et al., 2014). The solution is also similar, as it combines a declarative rule-based approach with disambiguation and a back-up strategy.

3 The ud2gf basic algorithm

The basic algorithm is illustrated in Figure 4

Its main data-structure is an **annotated dependency tree**, where each node has the form

$< L, t, ts, C, p >$ where

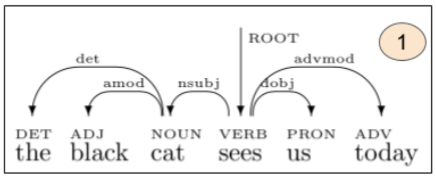
- L is a dependency label (always the same as in the original UD tree)
- t is the current GF abstract syntax tree (iteratively changed by the algorithm)
- ts is a list of alternative GF abstract syntax trees (iteratively changed by the algorithm)
- C is the GF category of t (iteratively changed by the algorithm)
- p is the position of the original word in the UD tree (always the same as in the original UD tree)

Examples of such nodes are shown in Figure 4, in the tree marked (5) and in all trees below it.

The algorithm works in the following steps, with references to Figure 4:

1. **Restructuring**. Convert the CoNLL graph (marked (2) in Figure 4) to a tree data-structure (3), where each node is labelled by a dependency label, lemma, POS tag, and word position. This step is simple and completely deterministic, provided that the graph is a well-formed tree; if it

Restructuring and lexical annotation



```
conll                                         2

 1  the    the     DET  _   3   det
 2  black  black   ADJ  _   3   amod
 3  cat    cat     NOUN _   4   nsubj
 4  sees   see     VERB _   0   root
 5  us     we      PRON _   4   dobj
 6  today  today   ADV  _   4   advmod
```

```
tree                      3

root see VERB _ 4

    nsubj cat NOUN _ 3

        det the DET _ 1

        amod black ADJ _ 2

    dobj we PRON _ 5

    advmod today ADV _ 6
```

```
lexicon              4

see_V2      “see”

cat_N       “cat”

the_Det     “the”

black_A     “black”

we_Pron     “we”

today_Adv   “today”
```

```
lexically annotated tree  5

root see_V2 V2 4

    nsubj cat_N N 3

        det the_Det Det 1

        amod black_A A 2

    dobj we_Pron Pron 5

    advmod today_Adv Adv 6
```

A node annotation by endo- and exocentric functions

```
tree


root see_V2 V2 4

   nsubj (UseN 3) [cat_N] CN 3

      det the_Det Det 1

      amod (PositA 2) [black_A] AP 2

   dobj (UsePron 5) [we_Pron] NP 5

   advmod today_Adv Adv 6
```

```
endo



ModCN 2 3
```

```
exo



DetCN 1 3
```

The final annotated tree

```
root (PredVP 3 4) [(AdvVP 4 6),(ComplV2 4 5),see_V2] VP 4

   nsubj (DetCN 1 3) [(ModCN 2 3),(UseN 3),cat_N] NP 3

      det the_Det Det 1

      amod (PositA 2) [black_A] AP 2

   dobj (UsePron 5) [we_Pron] NP 5

   advmod today_Adv Adv 6
```

Figure 4: Steps in ud2gf

isn't, the conversion fails[2].

2. **Lexical annotation**. Preserve the tree structure in (3) but change the structure of nodes to the one described above and shown in (5). This is done by using a GF lexicon (4), and a category configuration, replacing each lemma with a GF abstract syntax function and its POS with a GF category.[3]

3. **Syntactic annotation**. The GF trees t in the initial tree (5) are lexical (0-argument) functions. The syntactic annotation step annotates the tree recursively with applications of syntactic combination functions. Some of them may be **endofunctions** (i.e. **endocentric** functions), in the sense that some of the argument types is the same as the value type. In Figure 3, the functions `AdvVP` and `ModCN` are endocentric. All other functions are **exofunctions** (i.e. **exocentric** functions), where none of the argument types is the same as the value type. In the syntactic annotation, it is important to apply endofunctions before exofunctions, because exofunctions could otherwise block later applications of endofunctions.[4] The algorithm is a depth-first postorder traversal: for an annotated tree $T = (N T_1 \ldots T_n)$, where $N =< L,t,ts,C,p >$,

- syntax-annotate the subtrees $T_1,\ldots,T_n$
- apply available combination functions to N:
  - if an endofunction $f : C \to C$ applies, replace $< t,ts >$ with $< ((f t), \{t\} \cup ts >$
  - else, if an exofunction $f : C \to C'$ applies, replace $< t,ts,C >$ with $< (f t), \{t\} \cup ts, C' >$

where a function $f : A \to B$ **applies** if $f = (\lambda x)(g \ldots x \ldots)$ where g is an endo- or exocentric function on C and all other argument places than x are filled with GF trees from the subtrees of T. Every subtree can be used at most once.

An example of syntactic annotation is shown in the middle part of Figure 4. The node for the word *cat* at position 3 (the second line in the tree) has one applicable endofunction, `ModCN` (adjectival modification), and one exofunction, `DetCN` (determination). Hence the application of the endofunction `ModCN` combines the `AP` in position 2 with the `CN` in position 3. For brevity, the subtrees that the functions can apply to are marked by the position numbers. [5] Hence the tree

```
DetCN 1 3
```

in the final annotated tree actually expands to

```
DetCN the_Det
  (ModCN (PositA black_A) (UseN cat_N))
```

by following these links. The whole GF tree at the root node expands to the tree shown in Figures 2 and 3.

4 Refinements of the basic algorithm

We noted in Section 2 that ud2gf has to deal with ambiguity, incompleteness, noise, and ungrammaticality. The basic algorithm of Section 3 takes none of these aspects into account. But it does contain what is needed for ambiguity: the list ts of previous trees at each node can also be used more generally for storing alternative trees. The "main" tree t is then compared and ranked together with these candidates. Ranking based on tree probabilities in previous GF treebanks, as in (Angelov, 2011), is readily available. But an even more important criterion is the **node coverage** of the tree. This means penalizing heavily those trees that don't cover all nodes in the subtrees.

This leads us to the problem of incompleteness: what happens if the application of all possible candidate functions and trees still does not lead to a tree covering all nodes? An important part of this problem is due to **syncategorematic words**. For instance, the copula in GF is usually introduced as a part of the linearization, and does not have a category or function of its own.[6] To take the simplest possible example, consider the adjectival predication function and its linearization:

```
fun UseAP : AP -> VP
lin UseAP ap = \\agr => be agr ++ ap
```

where the agreement feature of the verb phrase is passed to an auxiliary function `be`, which produces the correct form of the copula when the subject is added. The sentence *the cat is black* has the following tree obtained from UD:

[2]This has never happened with the standard UD treebanks that we have worked with.

[3]The GF lexicon is obtained from the GF grammar by linearizing each lexical item (i.e. zero-place function) to the form that is used as the lemma in the UD treebank for the language in question.

[4]This is a simplifying assumption: a chain of two or more exofunctions could in theory bring us back to the same category as we started with.

[5]If the argument has the same node as the head (like 3 here), the position refers to the next-newest item on the list of trees.

[6]This is in (Kolachina and Ranta, 2016) motivated by cross-lingual considerations: there are languages that don't need copulas. In (Croft et al., 2017), the copula is defined as a **strategy**, which can be language-dependent, in contrast to **constructions**, which are language-independent. This distinction seems to correspond closely to concrete vs. abstract syntax in GF.

```
root (PredVP 2 4) [UseAP...black_A] S 4
  nsubj (DetCN 1 2) [UseN 2,cat_N] 2
    det the_Det Det 1
  cop "be" String 3 ***
```

The resulting GF tree is correct, but it does not cover node 3 containing the copula.[7] The problem is the same in gf2ud (Kolachina and Ranta, 2016), which introduces language-specific concrete annotations to endow syncategorematic words with UD labels. Thus the concrete annotation

```
UseAP head {"is","are","am"} cop head
```

specifies that the words *is,are,am* occurring in a tree linearized from a `UseAP` application have the label `cop` attached to the head.

In ud2gf, the treatment of the copula turned out to be simpler than in gf2ud. What we need is to postulate an abstract syntax category of copulas and a function that uses the copula. This function has the following type and configuration:

```
UseAP_ : Cop_ -> AP -> VP ; cop head
```

It is used in the basic algorithm in the same way as ordinary functions, but eliminated from the final tree by an explicit definition:

```
UseAP_ cop ap = UseAP ap
```

The copula is captured from the UD tree by applying a category configuration that has a condition about the lemma:[8]

```
Cop_ VERB lemma=be
```

This configuration is used at the lexical annotation phase, so that the last line of the tree for *the cat is black* becomes

```
  cop be Cop_ 3
```

Hence the final tree built for the sentence is

```
PredVP (DetCN the_Det (UseN cat_N))
  (UseAP_ be (PositA black_A))
```

which covers the entire UD tree. By applying the explicit definition of `UseAP_`, we obtain the standard GF tree

```
PredVP (DetCN the_Det (UseN cat_N))
  (UseAP (PositA black_A))
```

Many other syncategorematic words—such as negations, tense auxiliaries, infinitive marks—can be treated in a similar way. The eliminated constants are called **helper functions** and **helper categories**, and for clarity suffixed with underscores.

Another type of incomplete coverage is due to missing functions in the grammar, annotation errors, and actual grammar errors in the source text. To deal with these, we have introduced another type of extra functions: **backup functions**. These functions collect the uncovered nodes (marked with `***`) and attach them to their heads as adverbial modifiers. The nodes collected as backups are marked with single asterisks (`*`). In the evaluation statistics, they are counted as **uninterpreted nodes**, meaning that they are not covered with the standard GF grammar. But we have added linearization rules to them, so that they are for instance reproduced in translations. Figure 5 gives an example of a UD tree thus annotated, and the corresponding translations to Finnish and Swedish, as well as back to English. What has happened is that the temporal modifier formed from the bare noun phrase *next week* and labelled `nmod:tmod` has not found a matching rule in the configurations. The translations of the resulting backup string are shown in brackets.

5 First results

The ud2gf algorithm and annotations are tested using the UD treebanks (v1.4)[9]. The training section of the treebank was used to develop the annotations and the results are reported on the test section. We evaluated the performance in terms of coverage and interpretability of the GF trees derived from the translation. The coverage figures show the percentage of dependency nodes (or tokens) covered, and interpreted nodes show the percentage nodes covered in "normal" categories, that is, other than the Backup category. The percentage of interpreted nodes is calculated as the number of nodes in the tree that use a Backup function to cover all its children. Additionally, the GF trees can be translated back into strings using the concrete grammar, allowing for qualitative evaluation of the translations to the original and other languages.[10]

We performed experiments for three languages: English, Swedish and Finnish. Table 1 show the scores for the experiments using the gold UD

[7]We use `***` to mark uncovered nodes; since *be* has no corresponding item in the GF lexicon, its only possible categorization is as a `String` literal.

[8]The simplicity is due to the fact that the trees in the treebank are lemmatized, which means that we need not match with all forms of the copula.

[9]`https://github.com/UniversalDependencies/,retrievedinOctober2016`

[10]A quantitative evaluation would also be possible by standard machine translation metrics, but has not been done yet.

```
I have a change in plans next week .

root have_V2 : V2 2                      I have a change in plans "."
  nsubj i_Pron : Pron 1                  [ next week ]
  dobj change_N : N 4
    det IndefArt : Quant 3               minulla on muutos suunnitelmissa "."
    nmod plan_N : N 6                    [ seuraava viikko ]
      case in_Prep : Prep 5
  nmod:tmod Backup week_N : N 8 *        jag har en ändring i planer "."
    amod next_A : A 7 *                  [ nästa vecka ]
  punct "." : String 9
```

Figure 5: A tree from the UD English training treebank with lexical annotations and backups marked, and the resulting linearizations to English, Finnish, and Swedish.

language	#trees	#confs	%cov'd	%int'd
English	2077	31	94	72
Finnish	648	12	92	61
Finnish*	648	0	74	55
Swedish	1219	26	91	65
Swedish*	1219	0	75	57

Table 1: Coverage of nodes in each test set (L-ud-test.conllu). L* (Swedish*, Finnish*) is with language-independent configurations only. #conf's is the number of language-specific configurations. %cov'd and %int'd are the percentages of covered and interpreted nodes, respectively.

rule type	number
GF function (given)	346
GF category (given)	109
backup function	16
function config	128
category config	33
helper function	250
helper category*	26

Table 2: Estimating the size of the project: GF abstract syntax (as given in the resource grammar library) and its abstract and concrete configurations. Helper category definitions are the only genuinely language-dependent configurations, as they refer to lemmas.

trees. Also shown are the number of trees (i.e. sentences) in the test set for each language. The results show an incomplete coverage, as nodes are not yet completely covered by the available Backup functions. As a second thing, we see the impact of language-specific configurations (mostly defining helper categories for syncategorematic words) on the interpretability of GF trees. For example, in Swedish, just a small number of such categories (26) increases the coverage significantly. Further experiments also showed an average increase of 4-6% points in interpretability scores when out-of-vocabulary words were handled using additional functions based on the part-of-speech tags; in other words, more than 10% of uninterpreted nodes contained words not included in the available GF lexica.

Table 2 shows how much work was needed in the configurations. It shows the number of GF functions (excluding the lexical ones) and language-independent configurations. It reveals that there are many GF functions that are not yet reached by configurations, and which would be likely to increase the interpreted nodes. The helper categories in Table 2, such as Copula, typically refer to lemmas. These categories, even though they can be used in language-independent helper rules, become actually usable only if the language-specific configuration gives ways to construct them.

A high number of helper functions were needed to construct tensed verb phrases (VPS) covering all combinations of auxiliary verbs and negations in the three languages. This is not surprising given the different ways in which tenses are realized across languages. The extent to which these helper functions can be shared across languages depends on where the information is annotated in the UD tree and how uniform the annotations are; in English, Swedish, and Finnish, the compound tense systems are similar to each other, whereas negation mechanisms are quite different.

Modal verbs outside the tense system were another major issue in gf2ud (Kolachina and Ranta, 2016), but this issue has an easier solution in ud2gf. In GF resource grammars, modal verbs are a special case of VP-complement verbs (`VV`), which also contains non-modal verbs. The complementation function `ComplVV` hence needs two configurations:

```
ComplVV : VV->VP->VP ; head xcomp
ComplVV : VV->VP->VP ; aux  head
```

The first configuration is valid for the cases where the VP complement is marked using the `xcomp` label (e.g. *want to sleep*). The second one covers the cases where the VP complement is treated as the head and the VV is labelled `aux` (e.g. *must sleep*). The choice of which verbs are modal is language-specific. For example, the verb *want* marked as `VV` in GF is non-nodal in English but translated in Swedish as an auxiliary verb *vilja*. In gf2ud, modal verbs need non-local configurations, but in ud2gf, we handle them simply by using alternative configurations as shown above.

Another discrepancy across languages was found in the treatment of progressive verb phrases (e.g. *be reading*, Finnish *olla lukemassa*). in English the verb *be* is annotated as a child of the content verb with the `aux` label. In Finnish, however the equivalent verb *olla* is marked as the head and the content verb as the child with the `xcomp` label. This is a case of where the content word is not chosen to be the head, but the choice is more syntax-driven.

6 Conclusion

The main rationale of relating UD with GF is their complementary strengths. Generally speaking, UD strengths lie in parsing and GF strengths in generation. UD pipelines are robust and fast at analyzing large texts. GF on the other hand, allows for accurate generation in multiple languages apart from compositional semantics. This suggests pipelines where UD feeds GF.

In this paper, we have done preparatory work for such a pipeline. Most of the work can be done on a language-independent level of abstract syntax configurations. This brings us currently to around 70–75 % coverage of nodes, which applies automatically to new languages. A handful of language-specific configurations (mostly for syncategorematic words) increases the coverage to 90–95%. The configuration notation is generic and can be applied to any GF grammar and dependency scheme.

property	**UD**	**GF**
parser coverage	robust	brittle
parser speed	fast	slow
disambiguation	cont.-sensitive	context-free
semantics	loose	compositional
generation	?	accurate
new language	low-level work	high-level work

Table 3: Complementary strengths and weaknesses of GF and UD. UD strengths above the dividing line, GF strengths below.

Future work includes testing the pipeline in applications such as machine translation, abstractive summarization, logical form extraction, and treebank bootstrapping. A more theoretical line of work includes assessing the universality of current UD praxis following the ideas of Croft et al. (2017). In particular, their distinction between **constructions** and **strategies** seems to correspond to what we have implemented with shared vs. language-specific configurations, respectively.

Situations where a shared rule would be possible but the treebanks diverge, such as the treatment of VP-complement verbs and progressives (Section 5), would deserve closer inspection. Also an analysis of UD Version 2, which became available in the course of the project, would be in place, with the expectation that the differences between languages decrease.

Acknowledgements

We want to thank Joakim Nivre and the anonymous referees for helpful comments on the work. The project has been funded by the REMU project (Reliable Multilingual Digital Communication, Swedish Research Council 2012-5746).

References

Krasimir Angelov, Björn Bringert, and Aarne Ranta. 2014. Speech-enabled hybrid multilingual translation for mobile devices. In *Proceedings of the Demonstrations at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 41–44, Gothenburg, Sweden, April. Association for Computational Linguistics.

Krasimir Angelov. 2011. *The Mechanics of the Grammatical Framework*. Ph.D. thesis, Chalmers University of Technology.

William Croft, Dawn Nordquist, Katherine Looney, and Michael Regan. 2017. Linguistic Typology meets Universal Dependencies. In *Treebanks and Linguistic Theories (TLT-2017)*, pages 63–75, Bloomington IN, January 20–21.

Haskell B. Curry. 1961. Some Logical Aspects of Grammatical Structure. In *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society.

Prasanth Kolachina and Aarne Ranta. 2016. From Abstract Syntax to Universal Dependencies. *Linguistic Issues in Language Technology*, 13(2).

Peter Ljunglöf. 2004. *The Expressivity and Complexity of Grammatical Framework*. Ph.D. thesis, Department of Computing Science, Chalmers University of Technology and University of Gothenburg.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A Multilingual Treebank Collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, May. European Language Resources Association (ELRA).

Joakim Nivre. 2006. *Inductive Dependency Parsing*. Springer.

Aarne Ranta. 2004. Computational Semantics in Type Theory. *Mathematics and Social Sciences*, 165:31–57.

Aarne Ranta. 2009. The GF Resource Grammar Library. *Linguistic Issues in Language Technology*, 2(2).

Aarne Ranta. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.

Siva Reddy, Oscar Täckström, Michael Collins, Tom Kwiatkowski, Dipanjan Das, Mark Steedman, and Mirella Lapata. 2016. Transforming Dependency Structures to Logical Forms for Semantic Parsing. *Transactions of the Association for Computational Linguistics*, 4.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229.

Jörg Tiedemann and Zeljko Agic. 2016. Synthetic treebanking for cross-lingual dependency parsing. *The Journal of Artificial Intelligence Research (JAIR)*, 55:209–248.