Master thesis project planning report

Efficient conversion from Dependency Trees to Abstract Syntax Trees in Natural Language Processing

Andreas Källberg anka.213@gmail.com

Supervisor at CSE: Aarne Ranta

Relevant completed courses student 1:

TIN092 Algorithms
TDA342 Advanced functional programming
DAT151 Programming language technology

March 24, 2023

1 Introduction

In the beginning there was the word and it was good.

Applications: - Let a computer understand human language - Allow the computer to do things with human writing - For example, - - try to understan the meaning - - make transformations on the writing (which?) - - generate human language based on abstract knowledge

Reliablility and predictability are very important in many applications, for example law.

In rule based Natural Language Processing (NLP) we want to be able to take a text and analyse it at a higher level and make transformations to it. In order to do this, we first need a way to parse the text into a syntax tree, which describes the relationship between

There are several different formalisms for describing sentences as trees, all with their own strengths and weaknesses.

Something

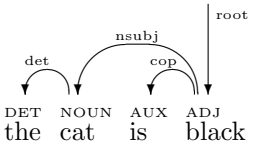
Write more about the goal of the project and move the rest to background

Something general about rule based NLG and NLP and computer grammars

1.1 Dependency trees and Universal Dependencies

A dependency tree is a tree structure that shows the grammatical relationship between words in a sentence. Each word becomes a node in the tree with the main verb as the root and the edges represents the relation and the direction of the dependency.

Add an image here



The specific standard for dependency trees that this paper is about is called Universal Dependencies (UD). UD is based on the idea of making a multilingual standard for dependency trees where the same set of tags can be used regardless of which language the sentence is written in.

Robust för ogrammatiska meningar osv och ger alltid ett resultat

Talk about the machine learning aspect of UD

1.2 Abstract trees and Grammatical Framework

Another way of describing the grammatical structure of a sentence is through abstract syntax trees, where instead of using words as the nodes in the tree, you use

Some basic intro on abstract trees

Write about abstract trees

Grammatical Framework[5] (GF) is a formalism for describing natural language grammars as code, which allows converting between natural language and a language-independent abstract syntax. It is split into a generic resource grammar that covers morphology and the language as a whole and application grammars for a more narrow domain which allows a more semantic abstract syntax. When you try to write a grammar that covers a very wide domain, you will often get an over-generating grammar where each sentence can be parsed in very many ways into many different trees, where usually only one is the intended way.

consider swapping order

1.3 Differences between GF and UD

GF is very strict when it comes to following grammar and spelling, which means that it will often refuse to parse sentences if they contain even the smallest error. UD on the other hand uses machine-learning to give some parse tree for all sentences regardless of how many errors they have.

While the machine-learning based approach of UD allows it to guess the correct tree better for ambiguous sentences and allows it to handle grammatically incorrect sentences better, GF is much more capable when it comes to performing transformations on the sentences, while maintaining correct morphology and grammar. This makes it attractive to parse sentences using UD and then convert the parsed trees to GF trees in order to perform further transformations.

2 Background

There exists a proof-of-concept implementation of a tool for converting between the trees for GF and UD, with the help of so called labels files which describe the mapping between UD labels and GF functions, called gf-ud, which contains both a component for converting from UD to GF, called ud2gf[7] and a component for converting from GF to UD, called gf2ud[2]. This work will focus on the ud2gf component.

these references apply to an older version of gf-ud, from before my work

gf-ud has been used for both translation and semantics[6]. Another application has been in concept alignment[4]. There has also been work using it for analysis of law text for the purpose of making a Controlled Natural Language for law[3].

actually explain this

```
1  cute  cute  ADJ  JJ  Degree=Pos  0  root  _  FUN=cute_A
2  ,  ,  PUNCT  ,  _  3  punct  _  _
3  fluffy  fluffy  ADJ  JJ  Degree=Pos  1  conj  _  FUN=fluffy_A
4  and  and  CCONJ  CC  _  5  cc  _  FUN=and_Conj
5  furry  furry  ADJ  JJ  Degree=Pos  1  conj  _  FUN=furry_A
```

Figure 1: The phrase "cute, fluffy and furry" as a textual UD tree

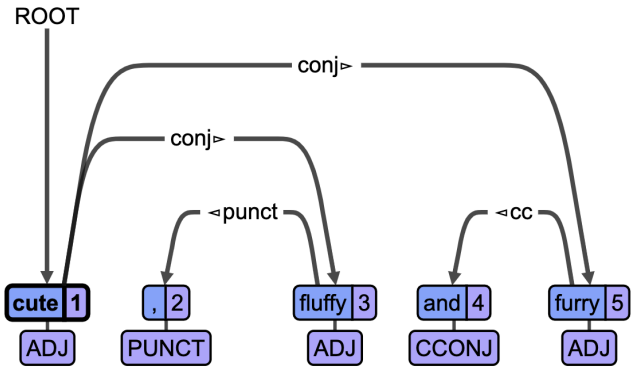


Figure 2: The phrase "cute, fluffy and furry" as a UD tree in graphical format

3 Problem

The current ud2gf implementation has some limitations. There are three main problems this work tries to fix.

The first problem is that it quickly becomes extremely slow for sentences more than a couple of words and/or when using large GF-grammars, e.g. GF-grammars containing Wordnet[1].
The second problem is that if the structure differs too much between the representation of a sentence in UD format and as a GF tree, it is not possible to describe the required transformation in the current "labels file" language. See section 3.1 below for more details.
The third problem is that it can sometimes be difficult to figure out why a rule in a labels file is not firing, so it would be useful to have a debugging tool to help diagnosing such issues.

3.1 Flexibility

As an example of a phrase that can be difficult to convert using the old gf2ud, let us consider the adjectival phrase "cute, fluffy and furry" would be described in UD format as in Figures 1 and 2.

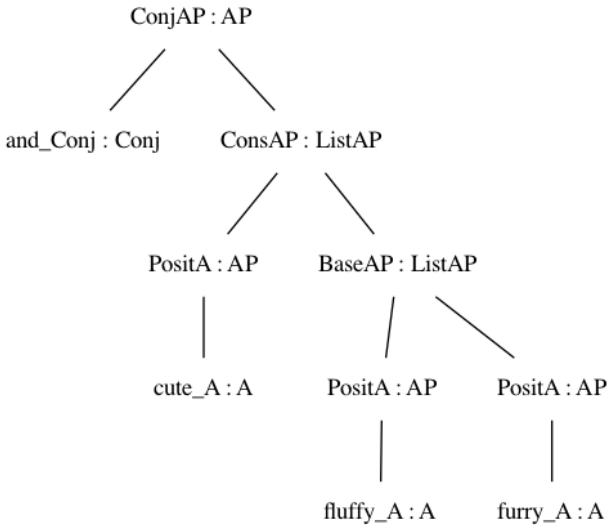


Figure 3: The phrase "cute, fluffy and furry" as a GF tree in graphical format.

The GF version of the same tree, shown in Figure 3, would look like this:

```
ConjAP and_Conj (ConsAP (PositA cute_A)
                        (BaseAP (PositA fluffy_A) (PositA furry_A)))
```

Here we can see that in UD, the word "cute" is in the root, while the conjunction "and" is at the bottom of the tree, while in GF the conjunction is a direct child of the root. This transformation can not be preformed by the simple single-layer transformations that are available in the current macro-system for labels files.

The translation described by a labels file is not one-to-one and there are often many possible GF trees that a UD tree could be translated to. The possible trees are currently ranked by completeness, as in how many of the words are included in the generated tree. However this ranking is incomplete and in case two possible trees, with the same GF category, cover the same words, an arbitrary tree will be chosen. A better choice could be to also check the linearization of these trees and rank those whose linearization is more similar to the original string higher. It would also be possible to completely exclude trees with differing linearization, but that would run counter to the goal of robustness.

4 Goals and Challenges

Change this so it's clear that it has already been done

1. Analyze algorithms and improve performance. The main challenge here is to find an algorithm for finding matching trees without exponential complexity on the number of children of a node in the UD tree.

2. Improve flexibility of macro language, allowing changing the structure of the trees while translating from GF to UD. One challenge here is in figuring out either how to change the algorithms to support these more advanced transformations or to find a way to allow them without needing to change the algorithms.

3. Write a debugging tool, which analyzes exactly what it is that prevents a rule in a labels file from firing or what prevents that tree from being selected. One challenge here is how to explain to the user the issue for all the possible things that can go wrong. There is also an engineering challenge in making an algorithm that figures out what went wrong and why.

4.1 Future work

4. If there is time, update the algorithm to look at the linearization in order to try to select trees that match the original string as closely as possible and evaluate what difference this makes. This is only possible with improved performance from goal 1, but keeping it fast will still be a challenge. There are also some challenges here in how it should interact with the advanced macros in goal 2. We also need to handle when a GF tree has multiple linearizations, e.g. if it results in a conjugation table.

5 Limitations

Only the direction of converting UD trees to GF trees is studied here, because that is what was relevant to the application at hand. Furhermore the two directions are almost completely independent in the implementation.

6 Approach

These methods were used for the different parts of the project

Change these to past tense

6.1 Performance

1. Finding the main source of slowness, which was done with profiling.

2. Analysing the current algorithms, which are based on brute-force, trying all combinations, with some simple filtering.

3. Finding a better algorithm, which avoids exploring paths that could never be the correct answer and which avoids duplicate work.

4. Analysing algorithmic complexity of both the new and the old algorithm and testing the practical performance to confirm the results.

6.2 Flexibility

In order to allow changing the shape of trees when translating from UD to GF, the macro language needs to be expanded. A first prototype of this can be done with minimal code changes by making macro expansion recursive and then representing the code for the transformation in Church-encoding, inspired by lambda-calculus.

This approach can be evaluated by seeing how well it covers different tree shape changes for different trees one would encounter.

It could also be worthwhile to make a more user-friendly version of the advanced macros that can be understood without knowing about Church-encoding

6.3 Debugging tool

Go through each component of the algorithms in order to find where applying a rule can go wrong and add detection for them. Additionally try out the debugging tool on a real grammar, e.g. in the context of [3], in order to find edge-cases which were not handled by the debugging tool.

6.4 Linearization-aware translation

Here we need a way to determine which linearization is actually closer and when to keep multiple options for a later stage of the translation. Some care also needs to be taken in determining which trees can be tossed away and which ones need to be kept around for a later stage.

Evaluating if the results are better with this version can be done in a large part by comparing the input string with the resulting string after translating to GF and linearizing.

7 Evaluation

One method for evaluation is through synthetic experiments of generating random GF trees through GF's built in functionality, then translating those to UD trees and then back to GF again. There are several different aspects that could be evaluated here:

- Completeness: the ability to always get complete trees without needing

to invent what GF2UD calls Backup GF funtions, for when no functions in the GF grammar was possible to use to connect a subtree

- Accuracy: How well the tree matches the original after a roundtrip
- Perfomance: How fast the code runs
- Error analysis: Examine the cause of errors and issues

8 Risk analysis and ethical considerations

There are very few risks to speak of as the practical work is already complete.

There are hardly any ethical considerations to speak of, since this is mostly a theoretical work. In some applications the reliablilty of the results can be critical, but in order to make such an application, the users would have to evaluate the reliablility as a part of the process. Additionally, being a rule-based system makes the results more predictable and makes it easier to fix any errors that comes up.

9 Time plan

Most of the practical implementation work has already been completed by the authors of this thesis prior to the official start of this thesis. The main parts that remain are the evaluation and writing the report itself. Optionally, if there is time, some additional work to further improve the results can be performed.

The plan is to complete the thesis before the summer of 2023.

Task	Target completion date
Implementation done	15th March
Halftime report	14th April
Evaluation experiment	28th April
Writing about evaluation	12th May
Finished presentation sent to Aarne	Two weeks before presentation
Presentation deadline	10th June

10 References

References

[1] K. Angelov and G. Lobanov. Predicting translation equivalents in linked wordnets. In *The 26th International Conference on Computational Linguistics (COLING 2016)*, page 26, 2016.

[2] P. Kolachina and A. Ranta. From Abstract Syntax to Universal Dependencies. *Linguistic Issues in Language Technology*, 13:1–57, 2016.

[3] Inari Listenmaa, Maryam Hanafiah, Regina Cheong, and Andreas Källberg. Towards CNL-based verbalization of computational contracts. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, Amsterdam, Netherlands, September 2021. Special Interest Group on Controlled Natural Language.

[4] Arianna Masciolini and Aarne Ranta. Grammar-based concept alignment for domain-specific machine translation. In *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, 2021.

[5] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[6] Aarne Ranta, Krasimir Angelov, Normunds Gruzitis, and Prasanth Kolachina. Abstract syntax as interlingua: Scaling up the grammatical framework from controlled languages to robust pipelines. *Computational Linguistics*, 46(2):425–486, 2020.

[7] Aarne Ranta and Prasanth Kolachina. From Universal Dependencies to Abstract Syntax. In *Proceedings of the 1st Workshop on Universal Dependencies*. Linköping University Electronic Press, May 2017.