Outsourcing morphology in Grammatical Framework: a case study for Hungarian

Inari Listenmaa

Department of Computer Science and Engineering
University of Gothenburg and Chalmers University of Technology
inari@chalmers.se

Abstract

We implement a miniature resource grammar in Grammatical Framework (GF) by using resources developed in the Apertium community: a finite-state morphological transducer and a disambiguation grammar. Our goals are twofold: to share resources within the rule-based community, as well as to prevent the GF grammar growing in size. Especially for languages with complex morphology, not having to store large inflection tables makes the grammar smaller and faster. As for development effort, we hope that the external resources would also save time in the grammar writing process. The next steps are to scale up to a full resource grammar, and parametrise the grammar for different tagsets.

1. Introduction

Grammatical Framework (GF, (Ranta, 2004)) is a grammar formalism and a programming language for writing multilingual grammars. A GF grammar consists of an *abstract syntax*, which declares the categories and constructions in the grammar, and a number of *concrete syntaxes*, where the categories and constructions are implemented, separately for each language. By allowing multiple concrete syntaxes for a single abstract syntax, GF is a natural choice for interlingual translation.

One of the most important contributions of GF is its Resource Grammar Library (Ranta, 2009), which contains 31 languages as of September 2016. All the languages share the same core abstract syntax, and each language can have an extra module for constructions that are particular to that language. Via the common core, we can translate basic syntactic structures between any pair of the 31 languages; any new language added to the library will be connected to all of the existing languages.

Typically, writing a morphological description is the first step in starting a resource grammar. A new grammar writer can easily spend weeks or months in defining inflection classes and writing morphophonological transformation rules for their language. Such a description in GF may well be valuable on its own right: it may provide insights about the morphological complexity of a language (Détrez and Ranta, 2012), or even lend itself for language description. However, creating a morphological description in GF is time-consuming, and the result is often less efficient than a finite-state description. If a new RGL language already has morphological resources, using them would ideally speed up both development and performance of the resource grammar.

2. Implementation

We implemented a miniature version (44 functions) of the GF resource grammar (Ranta, 2009) for this experiment. The development of this initial grammar took just a couple of hours; the author has no knowledge of Hungarian, but years of experience with GF and Apertium.

In the following sections, we present briefly the traditional way of writing GF grammars, followed by the new method.

2.1 Traditional GF grammar

A self-contained GF grammar stores inflection tables in each lexical entry. Then, the syntactic functions select the appropriate forms from the tables. We will illustrate the process with a grammar that has four categories: NP, V, VP and S, and two syntactic functions: complementation and predication. The `Compl` function combines a NP and a V into a VP, and `Pred` combines a NP and a VP into an S. The abstract syntax of this grammar is shown below.

```
abstract Grammar = {

cat
  S ; NP ; VP ; V ;

fun
  Pred : NP -> VP -> S ;
  Compl : V -> NP -> VP ;
}
```

Next, we write a concrete syntax for a language where verbs inflect for agreement and nouns for case. We introduce the parameters `Case` and `Agr`, and create the categories specific to this language.

```
param
  Case = Nom | Acc | Dat | ... ;
  Agr = SgP1 | SgP2 | ... | PlP3 ;
lincat
  S  = Str ;
  VP = Agr => Str ;
  V = { s : Agr => Str ; compl : Case } ;
  NP = { s : Case => Str ; agr : Agr } ;
```

Then we implement the functions which operate on the given categories. The complementation function must choose a right case from the object NP, depending on the verb. Likewise, the predication function must choose the right agreement from the VP, depending on the subject NP.

```
lin
  Compl v obj =
   table { agr => verb.s ! agr
               ++ obj.s ! v.compl } ;
```

```
    Pred subj vp = subj ! Nom
                ++ vp ! subj.agr ;
```

These parameters prevent the grammar from overgenerating. Given the NP *a fa* 'the tree' and the verb *szeret* 'love', the grammar will only accept *a fa szereti a fát* 'the tree loves the tree', no other combinations of cases and verb inflections.

On the high level, this grammar design looks simple. But under the hood, the GF source code is compiled into a low-level format. Adding a new value to a parameter or a new field to a record does not change much in the high-level rules, but each addition multiplies the number of low-level rules. As a result, GF grammars for morphologically complex languages often suffer from performance issues.

2.2 GF grammar with external resources

In the version with an external morphological analyser, all lexical entries contain only a base form, along with a POS tag. The features, which were used to select right forms from the inflection tables, are now replaced with string fields for *tags*, such as following.

```
  lincat
    S  = Str ;
    VP = { verb : Str ; obj : Str } ;
    NP = { s : Str ; agrTag : Str } ;
    V2 = { s : Str ; complTag : Str } ;
```

Then, the syntactic functions can be reduced into concatenating tags, and taking care of the right word order. Below are the same syntactic functions for complementation and predication.

```
  lin
    Compl v o = { verb = v.s ;
                  obj = o.s + v.complTag } ;

    Pred subj vp = (subj + "<nom>")
                ++ (vp.verb + subj.agrTag)
                ++ vp.obj ;
```

With this setup, the GF trees are linearised into what looks like Apertium analyses: for example, `mi<prn><pers><p1><mf><pl><nom>` `jár<vblex><past><p3><pl>` for 'we walked'.

2.3 Pipeline

The user of the grammar types in normal Hungarian words, and the input is analysed by the external morphological analyser, which is further disambiguated by a Constraint Grammar (Karlsson et al., 1995). Only then is the sentence given to the GF grammar, which will return the syntactic parse tree. The grammar can also be used in translation from another language to Hungarian. First the source language is analysed into a GF tree, then the tree is linearised into Hungarian with tags. To get actual Hungarian, the morphological analyser is called to generate the inflected forms from the tags.

Morphdb.hu	én/NOUN<PERS<1>><CAS<ACC>>
Apertium	én<prn><pers><p1><mf><sg><acc>

Table 1: Analyses for the pronoun *engem* 'me'

3. Discussion

Adding external tools into a GF grammar makes the system more complex. In comparison, a 100 % GF grammar is easier to understand, and immune to eventual changes in the other systems. If we only wanted to speed up development, we could use the FS morphology inside GF: generate all forms of the FS lexicon, and convert it into GF inflection tables. This solution would save the time in writing morphological rules, but not affect the size of the grammar. If there is already a GF morphology, we can also convert it into a FS morphology, via LEXC output from GF: this would aid performance, but not development.

There is another potential benefit of a GF grammar with tags: mapping from one tagset to another. To demonstrate, Table 3. shows two analyses of the Hungarian first person singular pronoun *engem* 'me', in Morphdb.hu (Trón et al., 2006) and Apertium. If we want to support both tagsets, we can write a second concrete syntax; or better yet, use the GF module system to parametrise over the concrete tags and the functions to combine them. The latter will require more work than the straight-forward solution shown in Section 2.2, but it is preferable to duplicating the work in syntax.

4. Conclusion

The experiment has been small, but promising in terms of effort. We estimate it would take days to write similar fragment in full GF, especially for a developer who doesn't know the language; in contrast, this grammar took merely hours to write. Evaluating the performance and correctness, as well as scaling up to a full resource grammar, is left for future work. In conclusion, we are happy with the results, and hope the experiment will encourage more sharing of resources within the rule-based community.

References

Grégoire Détrez and Aarne Ranta. 2012. Smart Paradigms and the Predictability and Complexity of Inflectional Morphology. In *EACL (European Association for Computational Linguistics)*, Avignon, April. Association for Computational Linguistics.

Fred Karlsson, Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1995. *Constraint Grammar: a language-independent system for parsing unrestricted text*, volume 4. Walter de Gruyter.

Aarne Ranta. 2004. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14:145–189.

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

Viktor Trón, Péter Halácsy, Péter Rebrus, András Rung, Péter Vajda, and Eszter Simon. 2006. Morphdb.hu: Hungarian lexical database and morphological grammar. In *Proceedings of 5th International Conference on Language Resources and Evaluation (LREC'06)*.