A Finite State Constraint Grammar Parser

Janne Peltonen
University of Helsinki
Helsinki, Finland
janne.peltonen@helsinki.fi

Abstract

It has long been held that finite state (FST) methods should be the method of choice in implementing a CG parser (e.g. Karlsson (1990)), since FST methods are very well understood mathematically and typically quite efficient. However, more or less all implementations to date have been using other methods. I set out to bridge the gap between the FST world and the CG world[1]. I created a representation for ambiguous, morphologically analysed sentences that might be general enough to be used in other projects as well. I was able to create a compilation procedure for most types of **CG-2** rules into an FST format, and successfully apply the compiled rules to my sentence representation. I implemented the grammar file parsing and rewrite rule generation using **Python 3**, and used Måns Huldén's (2009) **Foma** for the actual FST operations. I also evaluated the implementation in terms of time and space requirements.

1 Previous Work

My research is far from being the first attempt to combine the worlds of Constraint Grammars and finite state methods. For example, the *Finite State Intersection Grammars* (FSIG) developed by Koskenniemi (1990) are a decidedly CG-like finite state approach to disambiguation and surface syntactic parsing — so much so that FSIG has been suggested to be renamed Parallel CG and traditional CG, Sequential CG (Voutilainen, 1994). In FSIG, constraints act on complete sentences and prune complete readings on a sentence level, whereas traditional CG prunes readings from individual cohorts (word forms).

Gross defines *local grammars* loosely as a class of grammars that reduce ambiguity by local constraints (Gross, 1997). Mohri (2005) shows an algorithm to disambiguate an ambiguous sentence automaton using only local constraints. This, too, can be considered as a CG-like FST approach to disambiguation.

Graña *et al.* (2003) show a method to compile constraint-based textual rules directly to FSTs. However, their rule format differs somewhat from the previously used formats. My goal was to be backwards compatible with Tapanainen's **CG-2** (Tapanainen, 1996), since there are lots of grammars written using that formalism. I didn't use the open source **VISL CG-3** formalism since its differences from **CG-2** are minor — and the formalism is currently in a state of rapid evolution. Additionally, the Swahili grammar sample I was allowed to use in my work was written in **CG-2**.

2 Ambiguity Representation

One obvious problem to solve was the representation of sentences as finite state automata. There must be a representation for local ambiguity either (i) as word "lattice" containing paths that represent combinations of local readings; (ii) as a pearl chain shaped compressed word lattice where the dependencies across word boundaries are not maintained, but every combination have a separate path; or as (iii) as a single path automaton containing a string that lists all the local ambiguity classes (aka cohorts) on the single line.

The reasons for choosing the last methods — single path sentence automata — are, in no particular order, as follows:

- Avoiding the possibility of exponential search times.

[1]The results presented herein are also published in my master's thesis in Finnish in June, 2011.

- Apparent straightforwardness of composing a single path automaton to the rule automata.
- Possibility to refer to sibling readings, that is, readings in the same cohort.
- Usability of the representation even if the implementation of rules differs radically from the currently adopted one.

The actual form I chose is as follows:

- Cohorts are separated by the symbol ¤:
  `¤ cohort ¤ cohort ¤ ... ¤ cohort ¤`
- Readings are separated by the symbol §:
  `§ reading § reading § ... § reading §`
- There is a cohort separator at the beginning and end of the sentence, as well as a reading separator before the first reading and after the last one
- The word form is between the cohort separator and the first reading:
  `¤ "<word-form>" § reading § ...`
- The base form is the first tag in the reading:
  `... § "base-form" TAG1 TAG2 ...`
- The word form is repeated as the last tag in each reading:
  `... § "base-form" TAG1 TAG2 ... "<word-form>" §`

Figure 2 shows the sentence automaton for two word forms (cohorts) in an ambiguous sentence. In the traditional vertical form, the sentence would appear as in example 1.

(1)
```
...
"<sm1>"
        "pm11" P111
        "pm12" P121 P122
"<sm2>
        "pm21" P211
        "pm22" P221
...
```

3 Rule Representation

A natural form to use for the CG disambiguation rules themselves was Lauri Karttunen's Replace rule syntax (Karttunen, 1995) — there was an existing, open implementation available, and the rule formalism appeared strong and simple enough. The only problem with replace rules were **CG-2**'s (Tapanainen, 1996) linked rules: theoretically, a linked rule's left context might span to the right side of the rule target, and there is no easy or easily generalisable way to represent a left context that might leak to the right side of the rule centre in Karttunen's formalism. I chose to treat linked rules as a special case treated later. In the test grammar I have available there were only four linked rules, so that limitation appeared to be within reason.

I wanted the application strategy of the rules to mirror what I understood of current **CG-2** parsers as closely as possible. That is, at the rule level, I wanted the rules to appear to proceed from left to right — the left context of a given rule had to have the appearance of the rule being already applied there whereas the right context should be uncharted territory (Tapanainen, 1996). So the obvious variant of Karttunen's rules was the right-oriented (`//`) one — left context from the lower, or output, tape; right context from the upper, or input, tape.

The choice of right-oriented replace rules created interesting complications in the replace rules, especially when combined with the robustness clause 'the last reading of a cohort shall not be removed'. To elaborate: if a left context is to apply, the readings that contain the left context must not be marked as erased — or, in the case of a negated left context, they must all be marked as erased. But if all the readings in the cohort are marked as erased, then the last reading of the cohort should be treated as non-erased after all, because there has to be at least one reading left in each cohort. To be consistent with the left-to-right application strategy, that should be the rightmost one.

A simple first approach for rule 2 would be as in example 3. Here, a `REMOVEREADING` tag is added next to the target tag if there is a context tag `CTAG` in the previous cohort. `.¤.` means all strings that contain at most one cohort separator.

(2) `REMOVE (TTAG) IF (-1 (CTAG));`

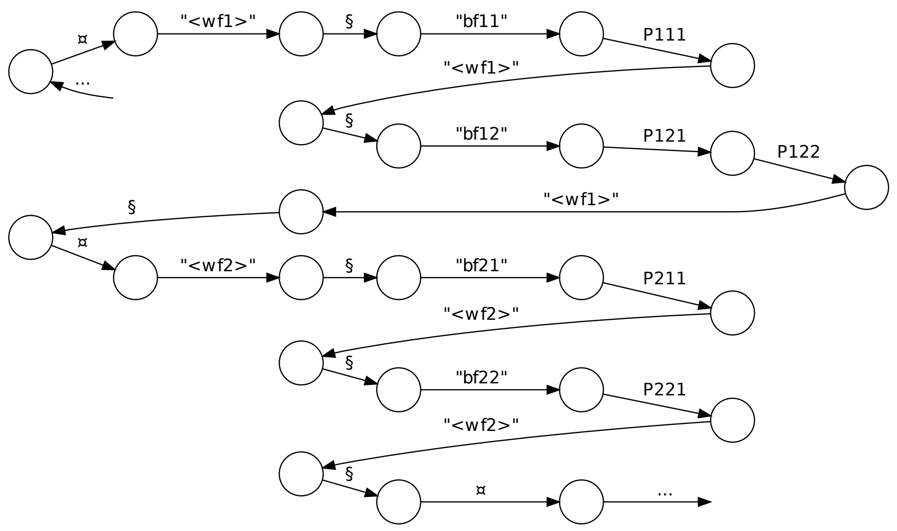

Figure 1: A Single Path Sentence Automaton

```
(3) TTAG -> TTAG REMOVEREADING ||
      CTAG .¤. _ ;
```

However, rule 3 doesn't reflect the internal rule application strategy outlined. To achieve that, a more complicated approach is needed. The end result is represented as rule 4.

```
(4) TTAG -> TTAG REMOVEREADING //
      .#. ... [
       ¤ ..
        § .nRR CTAG .nRR §
         .. |
       ¤ .
        [§ . REMOVEREADING .]*
         § . CTAG . §
      ] ¤. _ ;
```

The left context of rule 4 is composed as follows:

- `.#. ...` anchors the context to the beginning of the sentence. This is needed when combining context constraints and wouldn't be strictly necessary in this rule, with only one constraint.
- Two possibilities for the context cohort follow:
  1. The cohort contains a reading with the context tag `CTAG` and no `REMOVEREADING` tags (`.nRR` matches everything in a reading except a `REMOVEREADING` symbol); or
  2. all the previous readings in the cohort are marked as removed, so we don't care if the last reading is marked as removed — if it contains a `CTAG`, it matches.
- `¤.` means one immediate cohort separator and anything inside a cohort after that.

On higher level, that is, the levels of rule sets and grammar files, I chose to mimic the rule application order of Pasi Tapanainen's **CG-2** implementation (Tapanainen, 1996). First, a working set of rules is generated from the first constraint section in the grammar file, in the order they appear in the grammar file[2]. Then, for the sentence being processed, each rule is applied in isolation, and a new sentence automaton is created from the result of rule application — if there was a change; otherwise, the original sentence automaton is used with the next rule. When all the rules in the working set have been applied once, the whole process is repeated, until the working set of rules can no longer change the sentence automaton. At this point, the rules from the next constraint section in the grammar file are appended to the end of the working set, and the process is repeated until there are no more constraint sections left.

Changing the application order of rules would be relatively easy, since the rule application order

[2]This is actually different from Tapanainen's implementation which gives no guarantees about the application order of rules inside a constraint section; in Tapanainen's implementation, only the application order of constraint sections is defined to match the order of the sections in the grammar file.

is mostly defined in **Python**. On the other hand, running multiple rules in parallel for one sentence — in essence, trying to disambiguate each cohort in a sentence with all rules before advancing to the next cohort — is not easily achievable with my approach. It could be doable, in theory, by combining the context constraints of all the rules to create one huge replace rule. However, very complex replace rules appear to be slow to handle.

4 Implementation

The CG rule parser/re-writer was written in **Python 3**[3], using the PLY (**Python Lex-Yacc**) parser generator[4]. Måns Huldén's **Foma**, as a sub-process of the **Python** script, is used to compile the replace rules into transducers. The conversion between the traditional CG sentence format and the new sentence format is done in pure **Python**. **Foma**, as a sub-process of the **Python** script that controls the high level application order of rules, is also used to apply the rules to sentences.

To create the compilation process from CG rules to rewrite rules, I went through the different disambiguation rule types in Pasi Tapanainen's **CG-2** version of the CG formalism and came up with a rewrite rule equivalent for each of them. In the current implementation, the rewrite rule generation phase isn't as elegant as it could be: the rule generator goes through all different combinations of **CG-2** rule features and creates a rule for each combination separately, even if some of the features could be treated as modifiers of the original rule. For example, I suspect that I could implement rule negation as a simple textual transformation of the positive version of the same rule.

Using **Foma** with **Python** proved to be quite simple: the sub-process modules in **Python**'s standard library provided me with sufficient means to create a sub-process for **Foma** and communicate with it. Currently, I only have a simple implementation that reads lines from the **Foma** sub-process until a certain known line is reached, but I've been experimenting with a separate thread for communication, to avoid accidental lockups. So far, the results have been encouraging and simple to implement.

The implementation, in its current form, is not really distributable. I plan to create a distributable package and make it available on-line soon. In the meantime, I can provide the interested with the current version and instructions on how to make it function.

[3]`http://www.python.org/download/releases/3.0/`

[4]`http://www.dabeaz.com/ply/`

5 Test Grammar And Sentence Data

Professor emeritus Arvi Hurskainen allowed me to use a sample of his **SwaCG** Swahili language grammar to test and develop my own disambiguator (Hurskainen, 2004). The sample contains 397 rules of which 380 are select rules and 17 are remove rules; the approach used in this grammar was more to describe sufficient contexts for certain morphological choices than to describe when some readings should be discarded.

Four select rules contained a linked contextual test. My current implementation ignores rules with linked tests, so the results were, in effect, obtained with a grammar of 393 rules of which 376 are select rules and 17 remove rules.

The morphologically analysed test sentences were also provided by professor Hurskainen. They are a set of 684 sentences, hand-crafted to test different aspects of the Swahili grammar. The sentences are categorised as follows:

- constructions with inflecting adjectives (287 sentences);
- constructions with uninflecting adjectives (133 sentences);
- demonstrative before noun, inflecting adjectives (183 sentences); and
- demonstrative before noun, uninflecting adjectives 81 sentences.

The ambiguous morphological analysis contains 5191 word forms and 11 455 readings, with punctuation included. With punctuation excluded, there are 3139 word forms and 9 403 readings.

6 Results

Rules are applied one at a time, so the rules don't have to worry about other rules interfering with their execution. This also applies when creating compositions of rules: conceptually, the upper tape of each new rule is the lower tape of the previous rule composed with previous rules and the sentence automaton, so the effect is the same as with applying the new rule to a new version of the sentence automaton.

6.1 Space Complexity

Memory requirements didn't appear to grow especially fast when composing more and more rules to the composition that begins with the sentence automaton. On the other hand, when trying to compose as few as two big rules into a composition rule without the sentence automaton, I could easily run out of memory on my workstation. For example, composing two rule transducers of sizes 2.7 KB and 21.2 KB — with two simple special transducers that actually erase the readings that are marked as erased, and clean up superfluous erase markers, between the rules — generates a transducer of size 341.8 KB. Composing rules of sizes 21.2 KB and 121.1 KB result in a transducer of size 4.5 MB, and trying to compose two rules of size 26.4 MB results in **Foma** finally running out of memory on my workstation, after having allocated more than 1.2 GB. 26.4 MB rules are rare, but at this growth rate, 26.4 MB rule compositions wouldn't be.

The sizes of the compressed transducer files, in **Foma** binary representation, vary between 852 B and 2.6 MB — there correspond to uncompressed transducers of sizes 668 B and 26.4 MB. The binary representation of the compiled grammar, with 379 compressed binary rules, takes 8.0 MB.

6.2 Time Complexity

Compilation of the test grammar of 393 rules took at most 90 s (plus 25 s for the packing and unpacking of the binary transducers). **VISL CG-3**'s **CG-2** compatibility mode is 1 500 times faster. On the other hand, the time was minutes, not hours, so it's almost usable.

Disambiguating the test set of 684 sentences created almost the same results as disambiguating with **VISL CG-3**. All the differences could be explained by the fact that **VISL CG-3**, as opposed to Tapanainen's **CG-2** parser and my program, does not take the order of tags into account neither in combined tags nor in same position tests with catenation. Moreover, **VISL CG-2** collapses multiple instances of same tag in a catenation into one (Tino Didriksen, personal discussion). This result is encouraging.

Less encouraging is the result that disambiguating the complete test set took 64 min 12 s. Again, **VISL CG-3** was 1 500 times faster. The reason for the apparent slowness of my approach is not completely clear. An obvious first guess would be overhead in inter-process communication or problems with the speed of the finite state tools used. As it turned out, there were a couple of issues. However, the results given in this chapter are obtained after having resolved most of the technical issues.

6.3 Attempts to Increase Speed

It appears that adding a new rule to a composition of the sentence automaton and other rules takes more or less the same time than composing the first rule to the sentence automaton, or perhaps slightly more. So it is possible to reduce the run time of the program by using longer rule compositions on each iteration, since the overhead of reading and writing data between processes decreases (since the number of iterations decreases). However, there appears to be a cutoff point of something like 20 rules per composition after which the decrease in overhead can no longer compete with the increase in composition time — at least with my test grammar and data.

One complication in communicating with **Foma** was that **libreadline**[5] calls sometimes took a really long time to complete — but without **libreadline**, I couldn't get the inter-process communication to work. A one line patch to **Foma**, to flush its standard output after the completion of a command, solved that problem, and the time to read a longish sentence representation into a **Foma** variable dropped from more than 200 ms with **libreadline** to approximately 40 ms without it.

There were a couple of memory leak issues within **Foma** that Måns Huldén was kind to fix more or less immediately. Also, he provided me with optimised versions of his replace rule translation formulae that sped up the grammar compilation considerably.

6.4 Analysis

The slow results given in section 6.2 are obtained after the modifications made in the last section. Thus it appears that at least currently, the tightest bottlenecks are elsewhere than in inter-process communication or tool errors.

According to the analysis I performed using the **Python cProfile** library[6], most of the time was actually spent composing the sentence and rule transducers together. This would indicate that

[5] `http://www.gnu.org/software/readline/`

[6] `http://docs.python.org/release/3.1.3/library/profile.html`

my program creates so complex replace rules that **Foma** can no longer handle them efficiently. As the most complex rule transducers have tens of thousands of states and nearly two million arcs, that is hardly surprising — the worst case has 32 219 states and 1 732 204 arcs. I have yet to find out why some rules are so complex and what, if anything, could be done to avoid such complexity.

7 Conclusion

In this paper, I have shown that it is possible to create a finite state CG implementation that is mostly compatible with **CG-2**. I was also able to create a useful finite state representation for the ambiguous sentences, to which the rules could be readily applied. Lauri Karttunen's replace rules proved to be a usable basis for representing CG rules. My implementation is not complete, and currently only disambiguates a grammar containing at most the rule types in my test grammar — to test the disambiguator with other grammars, the remaining rule types should be catered for.

However, even if I have provided a proof of concept implementation of a CG disambiguator, my program is too slow for any practical purposes. Additional research is called for. The program might be sped up a bit by replacing inter-process communication with direct library calls, once the required **Python** library bindings are in place. It might also be possible to simplify the generated replace rules. If these approaches fail, other finite state methods than replace rules should be considered. As indicated in personal discussion, at least Anssi Yli-Jyrä and Måns Huldén are currently planning such approaches.

References

Jorge Graña, Gloria Andrade, and Jesús Vilares. 2003. Compilation of constraint-based contextual rules for part-of-speech tagging into finite state transducers. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 128–137. Springer Berlin / Heidelberg.

Maurice Gross. 1997. The construction of local grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-state language processing*, pages 329–354. MIT, Cambridge (MA), USA.

Mans Hulden. 2009. Foma: a finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32, Athens, Greece, April. Association for Computational Linguistics.

Arvi Hurskainen. 2004. Optimizing disambiguation in swahili. In *Proceedings of Coling 2004*, pages 254–260, Geneva, Switzerland, Aug 23–Aug 27. COLING.

Fred Karlsson. 1990. Constraint grammar as a framework for parsing unrestricted test. In *COLING-90: papers presented to the 13th International Conference on Computational Linguistics: on the occasion of the 25th anniversary of COLING and the 350th anniversary of Helsinki University*, pages 168–173, Helsinki. International Conference on Computational Linguistics.

Lauri Karttunen. 1995. The replace operator. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 16–23, Morristown, NJ, USA. Association for Computational Linguistics.

Kimmo Koskenniemi. 1990. Finite-state parsing and disambiguation. In *COLING-90: papers presented to the 13th International Conference on Computational Linguistics: on the occasion of the 25th anniversary of COLING and the 350th anniversary of Helsinki University*, pages 229–232, Helsinki. International Conference on Computational Linguistics.

Mehryar Mohri. 2005. Local grammar algorithms. In Lauri Carlson, Antti Arppe, Mickael Suominen, Krister Lindén, Jussi Piitulainen, Martti Vainio, Hanna Westerlund, Anssi Yli-Jyrä, Juho Tupakka, and Markus Koljonen, editors, *Inquiries into Words, Constraints and Contexts. Festschrift for Kimmo Koskenniemi on his 60th Birthday*, CSLI Studies in Computational Linguistics, pages 84–94. CSLI Publications, Stanford, CA, USA.

Pasi Tapanainen. 1996. *The constraint grammar parser CG-2*. University of Helsinki, Department of General Linguistics, Helsinki.

Atro Voutilainen. 1994. *Designing a Parsing Grammar*. University of Helsinki, Department of General Linguistics, Helsinki.