Why implementation matters: Evaluation of an open-source constraint grammar parser

Dávid Márk Nemeskey
Institute for Computer Science and Control,
Hungarian Academy of Sciences,
H-1111 Budapest
nemeskey.david@sztaki.mta.hu

Francis M. Tyers
HSL-fakultetet,
UiT Norgga árktalaš universitehta,
9017 Romsa (Norway)
francis.tyers@uit.no

Mans Hulden
Department of Linguistics,
University of Colorado Boulder
80309 Boulder, Colorado
mans.hulden@colorado.edu

Abstract

In recent years, the problem of finite-state constraint grammar (CG) parsing has received renewed attention. Several compilers have been proposed to convert CG rules to finite-state transducers. While these formalisms serve their purpose as proofs of the concept, the performance of the generated transducers lags behind other CG implementations and taggers.

In this paper, we argue that the fault lies with using generic finite-state libraries, and not with the formalisms themselves. We present an open-source implementation that capitalises on the characteristics of CG rule application to improve execution time. On smaller grammars our implementation achieves performance comparable to the current open-source state of the art.

1 Introduction

Constraint grammar (CG), described originally by Karlsson (1990), is a rule-based formalism for various linguistics tasks, including morphological analysis, clause boundary detection and surface syntactic parsing. It has been used in a wide range of application areas, such as morphological disambiguation, grammar checking and machine translation (Bick, 2011). CG owns its popularity to two reasons: first, it achieves high accuracy on free text. Second, it works for languages where the annotated corpora required by statistical parsing methods are not available, but a linguist willing to work on the rules is. The original CG has since been superseded by CG-2 (Tapanainen, 1996) and lately, the free/open-source VISL CG-3 (Bick, 2000; Didriksen, 2011).

Constraint grammar, however, has its drawbacks, one of which is speed. The Apertium machine translation project (Forcada et al., 2011) uses both CG (via VISL CG-3) and n-gram based models for morphological disambiguation, and while CG achieves higher accuracy, the n-gram model runs about ten times faster.

In this paper, we investigate how using finite-state transducers (FST) for CG application can help to bridge the performance gap. In recent years, several methods have been proposed for compiling a CG to FST and applying it on text: Hulden (2011) compiles CG rules to transducers and runs them on the input sentences; Peltonen (2011) converts the sentences into ambiguous automata and attempts to eliminate branches by intersecting them with the rule FSTs; finally, Yli-Jyrä (2011) creates a single FST from the grammar and applies it on featurised input. Unfortunately, none of the authors report exact performance measurements of their systems. Yli-Jyrä published promising numbers for the preprocessing step, but nothing on the overall performance. Peltonen, on the other hand, observed that "VISL CG-3 was 1,500 times faster" than his implementation (Peltonen, 2011).

Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, pages 772–780, Dublin, Ireland, August 23-29 2014.

We do not attempt here to add a new method to this list; instead, we concentrate on three practical aspects of FST-based CG. First, we report accurate measurements of the real-world performance of one of the methods above. Second, we endeavour to optimise the implementation of the selected method. All three works used *foma*, an open source FST library (Hulden, 2009b; Hulden, 2009a). We show that while *foma* is fast, relying on specialised FST application code instead of a generic library clearly benefits performance. We also demonstrate what further improvements can be achieved by exploiting the peculiarities of CG. Lastly, our research also aims to fill the niche left by the lack of openly accessible finite-state CG implementations.

Section 2 briefly introduces the method we chose to evaluate. In the rest of the paper, we present our optimisations in a way that mirrors the actual development process. We start out with a simple rule engine based on foma, and improve it step-by-step, benchmarking its performance after each modification, instead of a single evaluation chapter. We start in Section 3 by describing our evaluation methodology. Section 4 follows the evolution of the rule engine, as it improves in terms of speed. Section 5 contains a complexity analysis and introduces an idea that theoretically allows us to improve the average- and best-case asymptotic bound. Section 6 demonstrates how memory savings can be derived from the steps taken in section 4. Finally, Section 7 contains our conclusions and lists the problems that remain for future work.

2 The *fomacg* compiler and *fomacg-proc*

We have chosen Hulden's *fomacg* compiler for our study. Our reasons for this are twofold. The transducers generated by *fomacg* were meant to be run on the input directly, but they could also be applied to a finite-state automaton (FSA) representation of the input sentence via FST composition, thereby giving us more space to experiment. Peltonen's method, on the other hand, works only through FST intersection. More importantly, *fomacg* was the only compiler that is openly available.[1]

Here we briefly describe how *fomacg* works; for further details refer to (Hulden, 2011). A CG used for morphological disambiguation takes as input a morphologically analysed text, which consists of *cohorts*: a word with its possible readings. A reading is represented by a lemma and a set of morphosyntactic *tags*. For example, the cohort of the ambiguous Hungarian word *szív* with two readings "heart" and "to suck" would be `^szív/szív<n><sg><nom>/szív<vblex><pres><s3p>$`.[2] The text is tokenised into sentences based on a set of *delimiters*. CG rules operate on a sentence, removing readings from cohorts based on their context. The rules can be divided into priority levels called *section*. Most implementations apply the rules one-by-one in a loop, until no rules can further modify the sentence.

fomacg expects cohorts to be encoded in a different format; the cohort in the example above would be represented as

```
$0$ "<szív>" #BOC#         |
#0# "szív" n sg nom        |
#0# "szív" vblex pres s3p  | #EOC#
```

The rule transducers mark readings for removal by replacing the `#0#` in front of the reading by `#X#`; they act as identity for sentences they cannot be applied to.

fomacg is only a compiler, which reads a CG rule file and emits a *foma* FST for each rule. The actual disambiguator program that applies the transducers to text we implemented ourselves. It reads the morphologically analysed input in the Apertium stream format, converts it into the format expected by *fomacg*, applies the transducers to it, and then converts the result back to the stream format. To emphasise its similarity to *cg-proc*, VISL CG's rule applier, we named our program *fomacg-proc*.

3 Methodology

Apertium includes constraint grammars for several languages.[3] While most of these are wide-coverage grammars, and are being actually used for morphological disambiguation in Apertium, they are also

[1] In the Apertium software repository: `https://svn.code.sf.net/p/apertium/svn/branches/fomacg`

[2] The example is in the Apertium stream format, not in CG-2 style.

[3] `http://wiki.apertium.org/wiki/Constraint_grammar`

too big and complex to be easily used for the early stages of parser development. Therefore, we have written a small Hungarian CG, aimed to fully disambiguate a short Hungarian story, which was used as the development corpus. Since Hungarian is not properly supported by Apertium yet, morphological analysis was carried out by Hunmorph (Trón et al., 2005), and the tags were translated to the Apertium tagset with a transducer in `foma`.

The performance of `fomacg-proc` has been measured against that of VISL CG. The programs were benchmarked with three Apertium CG grammars: the toy Hungarian grammar mentioned earlier, the Breton grammar from the `br-fr` language pair (Tyers, 2010), and the version of the Finnish grammar originally written by Karlsson in the North Sámi–Finnish (`sme-fin`) pair. Seeing that in the early phases, only the Hungarian grammar was used for development, results for the other two languages are reported only for the later steps.

Each grammar was run on a test corpus. For Breton, we used the corpus in the `br-fr` language pair, which consists of 1,161 sentences. There are no Finnish and Hungarian corpora in Apertium; for the former, we used a 1,620-sentence excerpt from the 2013-Nov-14 snapshot of the Finnish Wikipedia, while for the latter, the short test corpus used for grammar development. Since the latter contains a mere 11 sentences, it was repeated 32 times to produce a corpus similar in size to the other two.[4] The Breton and Finnish corpora were tagged by Apertium's morphological analyser tools.

Since VISL CG implements CG-3, and *fomacg* only supports CG-2, a one-to-one comparison with the grammars above was not feasible. Therefore, we extracted the subset of rules from each that compiled under *fomacg*, and carried out the tests on these subsets. Table 1 shows the number of rules in the original and the CG-2 grammars.

Table 1: Grammar sizes with the running time and binary size of the respective VISL-CG grammars

Language	Rules	CG-2 rules	Binary	Time
Hungarian	33	33	8kB	0.284s
Breton	251	226	36kB	0.77s
Finnish	1207	1172	184kB	1.78s

We recorded both initialisation and rule application time for the two programs, via instrumentation in case of *fomacg-proc* and by running the grammar first on an empty file and then on the test corpus in case of *cg-proc*. However, as initialisation is a one-time cost, in the following we are mainly concerned with the time required for applying rules. The tests were conducted on a consumer-grade laptop with a 2.2GHz Core2Duo CPU and 4GB RAM, running Linux.

4 Performance optimisations

Our implementation, much like that of *fomacg* (and indeed, all recent work on finite state CG) is based on the *foma* library. We started out with a naïve implementation that used solely stock *foma* functions. Most of the improvements below stem from the fact that we have replaced these functions with custom versions that run much faster. The final implementation abandons *foma* entirely, but for the data structures. In the future, we plan to discard those as well, making our code self-contained.

The program loads the transducers produced by *fomacg* and applies them to the text. The input is in the Apertium stream format[5] and it is read cohort-by-cohort. A *foma* FST is used to convert each cohort to the format expected by the rule transducers, and to convert the final result back.

To tokenise the text to sentences, we modified *fomacg* to compile the *delimiters* set and emit it as the first FSA in the binary representation of the grammar. *fomacg-proc* reads the input until a cohort matches this set and then sends the accumulated sentence to the rule applier engine.

[4] Although we used the same corpus for development and testing for Hungarian, the experimental setup was the same for VISL-CG and fomacg. While the numbers we acquired for Hungarian are not representative of how a proper Hungarian CG would perform on unseen data, they clearly show which of our steps benefit performance.

[5] `http://wiki.apertium.org/wiki/Apertium_stream_format`

The rules are tested one-by-one, section-by-section, to see if any of them can be applied to the text. Once such a rule is found, the associated FST is executed on the text. As it is possible that a rule that was not applicable to the original text would now run on the modified one, testing is restarted from the first section after each rule application. The process ends when no more applicable rules are found.

4.1 Naïve implementation

The first version of the program used the `apply_down()` *foma* function both for rule application and format conversion. As *fomacg* generated a single FST for a rule, rule testing and execution was done in the same step, by applying the FST. Whether the rule was actually applied or not was decided by comparing the original sentence to the one returned by the function.

The first row in Table 2 shows the running time for the Hungarian grammar. At 6.4s, the naïve implementation runs more than 20 times slower than VISL-CG (see Table 1). Luckily a far cry from the 1,500 reported by Peltonen, but clearly too slow to be of practical use.

4.2 FST composition

Another way to apply a rule is to convert the input sentence into a single-path FSA with the same alphabet as the rules and compose the rule FST on top of it. To check if the rule has actually be applied, the input automaton was intersected with the result. Unfortunately, this method proved to be much slower than the application-based one; composition alone took 28.3 seconds on our corpus, while the intersection pushed it up to 45s. Therefore we decided to abandon this path altogether.

4.3 Deletion of discarded readings

The original transducers replace the `#0#` in front of discarded readings with `#X#`. Our first optimisation comes from the observation that deleting these readings instead would not make the transducers any more complex, but would shorten the resulting sentence, making subsequent tests faster. Moreover, it allows the engine to recognise actual rule application by simply testing the length of the output to the input sentence, an operation slightly faster than byte-for-byte comparison.

Table 2 reports an approximately 8% improvement. While not self-evident, this benefit remained in effect after our subsequent optimisations.

4.4 FSA-based rule testing

Theoretically, further speed-ups could be achieved by separating rule testing and application, using finite-state automata for the former. Automata are faster than transducers for two reasons: first, there is no need to assemble an output; and second, a FSA can be determinised and minimised, while *foma* can only make a FST deterministic by treating it as a FSA with an alphabet of the original input:output pairs, which does not entail determinism in the input.

As the fourth row in table 2 shows, the idea does not immediately translate well to practice. The fault lies with the `apply_down()` function, which, being the only method of running a finite-state machine in *foma*, was designed to support all features of the library. It treats automata as identity transducers, and fails to capitalise on the aforementioned advantages of the former. In order to benefit from FSA-based testing then, a custom function is required.

4.5 Custom FSA/FST application

The `apply_down()` function supports the following features (Hulden, 2009a):

- Conversion of the text to symbols (single- and multi-character)
- Regular transitions and flag diacritics
- Three types of search in the transition matrix (linear, binary and indexed)
- Deterministic and non-deterministic operation
- Iterators (multiple invocations iterate the non-deterministic outputs)

Our use-case makes most of these features surplus to requirements. *fomacg* uses multi-character symbols, but not flag diacritics. To maximise the performance gains, the rule testing automata must be minimal (hence deterministic), so there was no need for non-determinism and iterators. Finally, by modifying *fomacg* to sort the edges of all grammar machines, we could ensure that binary transition search alone suffices.

The custom FSA applier function that implements only the necessary features was employed for both rule testing and finding the delimiter cohort. As a result, running time went down to 1.45 seconds (see table 2), a 75% improvement.

A similar function was written for input-deterministic minimal transducers. While not applicable to the non-deterministic rule FSTs, it could replace `apply_down()` for the conversion between the Apertium and the *fomacg* formats, further reducing the running time to 1.275 seconds.

What we can take home from the last two sections is that when speed is paramount, relying blindly on generic libraries may not only lead to suboptimal performance, but may also produce counterintuitive results.

Conversely, libraries may benefit from including specialised implementations for different use-cases. For example, *foma* has all the information at hand to decide if a FST is deterministic, whether it supports binary search or not, etc. and so, providing specialised functions (even private ones hidden behind `apply_down()`) would improve its performance substantially in certain situations.

4.6 Exploiting CG structure

In this chapter, we review the improvements made available by the characteristics of our CG representation. The first of these is functionality: even though the rule FSTs are non-deterministic, the input-output mapping is one-to-one (Hulden, 2011). It was thus possible to implement the non-deterministic version of the FST runner function described in the last section without the need for an iterator feature, and to use it for rule application. The last usage of the generic `apply_down()` function thus eradicated, the running time dropped to 1.05 seconds (see table 2).

Internally *foma*, similarly to other FST toolkits, represents elements of the Σ alphabet as integers. The conversion of text into tokens in Σ is a step usually taken for granted in the literature, but it contributes to the execution time of an FST to a significant extent. In *foma*, token matching is performed by a trie built from the symbols in the automaton's alphabet. Our custom DFSA runner function (see section 4.5) spends about 60% of its time applying this trie.

The two enhancements below have helped to all but negate the cost of token conversion. The first of these exploits the fact that in the `fomacg` format, symbols are separated by space characters. Instead of passing the input string to each FSM, we split it along the spaces, and pass the resulting string vector to the machines. This is a rather small change, and while the Hungarian grammar benefited almost nothing, the running time of the Breton grammar improved by 40%.

The second enhancement came from the observation that all rule testing automata and rule transducers accept the same CG tags. It is thus possible to generate an automaton whose alphabet is the union of those of the other machines. This automaton could be used to convert the input sentence into a vector of Σ ids, and then this vector could be sent to the other machines, relinquishing the need of repeated conversions.

Both *fomacg* and *fomacg-proc* had to be modified to account for the changes. The former now creates the converter FSA and saves it as the second machine in the binary grammar file. Also, since the ids that correspond to a symbol are unique to each machine, we added a post-processing phase that replaces the ids with the "canonical" ones in the converter FSA. *fomacg-proc* then converts the input to ids using the converter automaton's trie, and sends the vector to the rule machines. The rule machines treat the vector as their input, with a caveat: ids not in the alphabet of the machine in question are replaced by `@IDENTITY_SYMBOL@`, so that they are handled in the same way as before.

Table 2 shows that factoring the symbol conversion out from the individual machines resulted in huge savings: the running time of the Hungarian setup improved by 70% to 0.32 second; the Breton one by 40% to 1.55 seconds.

Table 2: Effects of the optimisations on running time

Version	Hungarian	Breton
Naïve (4.1)	6.4s	–
Composition (4.2)	45s	–
Delete readings (4.3)	5.9s	–
FSA rule testing (4.4)	10s	–
Custom FSA runner (4.5)	1.45s	–
Custom format-FST (4.5)	1.275s	6.8s
Input partitioning (4.6)	1.15s	4s
Custom rule applier (4.6)	1.05s	2.6s
One-time conversion (4.6)	0.32s	1.55s

5 Complexity analysis

Tapanainen (1999) proves that the worst-case time complexity for disambiguating a sentence in his CG-2 parser is $\mathcal{O}(n^3k^2G)$, where n is the length of the sentence, k is the maximum number of readings per word, and the grammar consists of G rules. The explanation is as follows: testing a cohort with a single rule can be done in $\mathcal{O}(nk)$; the whole sentence in $\mathcal{O}(n^2k)$. This process must be repeated for each rule, yielding $\mathcal{O}(n^2kG)$. Finally, in the worst case, a rule only removes a single reading, so it takes $n(k-1)$ rounds to disambiguate the sentence, resulting in the aforementioned bound.

Hulden (2011) showed that if the rules are compiled to transducers, they can be applied to the whole sentence in $\mathcal{O}(nk)$ time, thus decreasing the complexity to $O(n^2k^2G)$, instead of the $O(n^2k)$ suggested by Tapanainen. To be more precise, applying a rule transducer takes $O(nkT)$ time, where the constant T is the size of the FST. While T may be rather large, rule transducers may be factored into bimachines, which removes the constant. Hence, a disambiguating bimachine for one CG rule can be applied to a sentence of nk tokens in $O(nk)$ (linear) time. However, *fomacg* only includes CG rule-to-transducer compilation and does not include bimachine factorization as of yet.

While this work has left the theoretical limit untouched thus far, it improved on three aspects of the complexity. First, unlike *foma*, our specialised FST application functions can take advantage of the properties of automata and bimachines, and actually run them in $\mathcal{O}(nk)$ time. Second, the constant in the $\mathcal{O}$ has been decreased as a result of extensive optimisation. Third, rule testing automata have been introduced which, being minimal, can also be applied in $\mathcal{O}(nk)$ time. Assume that in a round G_a rules can be applied to the sentence and G_u cannot, $G_a + G_u = G$. With minimal automata for rule testing the round finishes with $2G_a + G_u$ machine applications, instead of the $2G$ required by bimachines. The facts that usually $G_a << G$ and that automata can be applied faster than transducers result in a performance improvement over the pure bimachine setup.

5.1 Beyond the $\mathcal{O}(n^2k^2G)$ bound

This section presents an idea that allows the system to theoretically overcome the $\mathcal{O}(n^2k^2G)$ average complexity bound. This section describes the method, and investigates its feasibility; the next section contains the evaluation.

The idea is based on the fact that regular languages are closed under the union operator. If there are two automata, FSA_{G_a} and FSA_{G_b} that test the rules G_a and G_b, respectively, then it follows that their union, $\mathrm{FSA}_{G_{ab}}$, accepts a sentence iff either G_a or G_b is applicable to it. If $\mathrm{FSA}_{G_{ab}}$ is minimised, it runs in $\mathcal{O}(nk)$ time, the same as FSA_{G_a} and FSA_{G_b}.

The union FSA allows us to implement hierarchical rule checking. In this example, testing if any of the two rules match a sentence with only the original automata requires a check with both. Instead, we can apply $\mathrm{FSA}_{G_{ab}}$ first. If neither rule is applicable, the automaton will not accept the sentence, and no further testing is required. If one of the rules is, FSA_{G_a} (or equivalently, FSA_{G_b}) must be run against the sentence to see which. In practice, if we pick two rules from a CG in random, we shall find that the majority of the sentences will not match either, hence the number of tests may be reduced substantially.

There is no need to stop here: we can take two union automata, and merge them again. It is easy to see that if we represent the rule testing automata in a graph, where a node is a FSA, and two nodes are connected iff one was created from the other via union, then we get a binary tree. For a grammar of G rules, a binary tree of $\log G$ levels can be built. Such a tree can confirm with a single test if a sentence does not match any of the rules, or find the matching rule in $\log G + 1$ tests, if one does. Accordingly, in theory this method allows us to improve the average- and best-case complexity bounds of the system to $\mathcal{O}(n^2k^2 \log G)$ and $\mathcal{O}(nk)$, respectively. (Clearly, for grammars with several sections, instead of a single tree that contains all rules, one tree must be built for each section to preserve rule priorities. However, this does not affect the reasoning above).

The bottleneck in this method is memory consumption. The size of the FSA resulting from a non-deterministic union operation is simply the sum of the sizes of the original automata. To achieve the speed-up described above, however, the rule checking automata must be determinised, which may cause them to blow up in size exponentially. Therefore, building a single tree from all rules is not feasible. A compromise solution is to construct a forest of 2–4 level trees, which still fits into the memory and provides similar benefits to a single tree, though to a smaller extent.

5.2 Evaluation

The forest can be assembled in several ways; we experimented with two simple algorithms. Both take as input a list of rule testing automata, which are encapsulated into single-node trees. Before each step, the trees are sorted by the size of the automata in their roots.

The first algorithm, *SmallestFirst*, unifies the two smallest trees in each step, until the root FSA in each tree is above a size limit (1,000 states in our case). The second, *FixedLevel*, aims to create full, balanced binary trees: in a single step, it unifies the smallest tree with the largest, the second smallest with the second largest, etc, and repeats the process until the trees reach a predefined height.

Table 3 lists the running times and memory requirements of the resulting forests. It can be seen that hierarchical rule testing indeed improves performance: even a single level of merging results in 30-42% speedup. However, it is also immediately evident that aside from special cases, the disadvantages overweight the benefits: memory usage and binary size grow exponentially, affecting compilation and grammar loading time as well, and very soon we run into the limits of physical memory. Unless a method is found that reduces memory usage substantially, we have to give up on hierarchical rule testing.

Table 3: Performance and storage requirements of rule testing trees
* State count limit was 500 † Reached limit of physical memory

Language	Algorithm	Initialisation	Disambiguation	Memory	File size
Hungarian	(flat)	0.028s	0.32s	0.5%	60kB
Hungarian	FixedLevel(3)	0.77s	0.235s	2.1%	7.1MB
Hungarian	Smallest First	0.62s	0.234s	1.9%	5.9MB
Breton	(flat)	0.5s	1.55s	5.1%	1.5MB
Breton	FixedLevel(2)	1.8s	1.09s	9.6%	7.4MB
Breton	Smallest First	11.14s	1.05s	28.7%	60MB
Finnish	(flat)	1.5s	22.87s	21.8%	7.2MB
Finnish	FixedLevel(2)	3.64s	13.28s	32.3%	28MB
Finnish	SmallestFirst*	20.75s	9.95s	–†	198MB

6 Memory savings

The use of a single converter automaton has not only resulted in improved performance, but it has also opened a way to decrease the storage space requirements of the grammar as well. The trie that converts the machine's alphabet to integer ids in *foma* takes up space; depending on the number and length of the symbols in bytes, this trie may be responsible for a considerable portion of the memory footprint of an

automaton. Given the number of rules in an average CG grammar, it is easy to see how this trivial subtask may affect the memory consumption of the application, as well as the size of the grammar binary. As the job of token matching has been delegated to the symbol automaton (see section 4.6), we no longer maintain separate tries for all individual FSAs.

Table 4 presents the resulting memory savings. We report numbers for the raw grammars (L1), as well as for two- and three-level condition trees (L2-3). It is not surprising that the raw grammars see the largest improvements; here the tries accounted for 70-80% of the memory usage. As the trees get higher, the number of states and edges grows more rapidly than does the number of tries and the savings become more modest.

Table 4: Improvements in memory usage due to removing the sigma trie. Memory consumption is measured as a percentage of the 4GB system memory

Language	Method	Before	After	Reduction
Hungarian	L1	0.5%	0.1%	80%
Hungarian	L3	2.1%	1.5%	28.57%
Breton	L1	5.1%	1.3%	74.5%
Breton	L2	9.6%	4.4%	54.16%
Finnish	L1	21%	4.1%	80.47%
Finnish	L2	32.3%	8.9%	72.44%

We explored other options as well to reduce the size of rule condition trees. Unfortunately, most methods aimed at FSA compression in the literature are either already implemented in *foma* (e.g. as row-indexed transition matrix, see Kiraz (2001)), or are aimed at automata with a regular structure, such as morphological analysers (Huet, 2003; Huet, 2005; Drobac et al., 2014). Without further support, the approximately 30% saving achieved by our method for a three-level condition tree alone is not enough to redeem hierarchical rule checking.

A task-specific framework, one based on inward deterministic automata has been proposed for CG parsing (Yli-Jyrä, 2011). The paper reports a binary size similar to the original grammar size. However, as the framework breaks away from the practice of direct rule application followed in this paper and in related literature (Hulden, 2011; Peltonen, 2011), closer inspection remains as future work.

7 Conclusions

We set out with the goal of creating a fast constraint grammar parser based on finite-state technology. Our aim was to achieve better performance on the task of morphological disambiguation than the current state-of-the-art parser VISL CG-3. We used the CG grammars available in the Apertium machine translation project.

Our goals were partially fulfilled: while the speed of our parser falls short of that of VISL CG-3 — with the exception of the execution of very small grammars — we have made advances on the state-of-the-art free/open-source FST implementations of CG. We based our system on the *fomacg* compiler, and extended it in several ways. Our parser uses optimised FST application methods instead of the generic *foma* variant used by previous implementations, thereby achieving better performance. Further optimisations, both memory and runtime, were made by exploiting the properties of FSTs generated from a CG. We report real-world performance measurements with and without these optimisations, so their efficacy can be accurately evaluated. A new method for rule testing has also been proposed, which in theory is capable of reducing the worst-case complexity bound of CG application to $\mathcal{O}(n^2k^2 \log G)$. Unfortunately, the method has yet to be proven feasible in practice.

Our main finding is that implementation matters: an FST library which is too generic hinders performance and can even make a theoretically faster algorithm slower in practice. Using bimachines and rule testing automata should have sped up rule application, but only did so after we implemented our own, specialised FST functions. Since *foma* has all necessary information about an FST in place to decide

the right application method, incorporating our functions into it, or other FST libraries, could benefit applications beyond the scope of CG.

Acknowledgements

This research was supported through an Apertium project in the 2013 Google Summer of Code.[6]

References

Eckhard Bick. 2000. *The parsing system" Palavras": Automatic grammatical analysis of Portuguese in a constraint grammar framework*. Aarhus Universitetsforlag.

Eckhard Bick. 2011. Constraint grammar applications. In *Proceedings of the NODALIDA 2011 Workshop: Constraint Grammar Applications*, page iv.

Tino Didriksen. 2011. Constraint grammar manual: 3rd version of the CG formalism variant.

Senka Drobac, Krister Lindén, Tommi Pirinen, and Miikka Silfverberg. 2014. Heuristic hyper-minimization of finite state lexicons. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland. European Language Resources Association (ELRA).

Mikel L Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M Tyers. 2011. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144.

Gérard Huet. 2003. Automata mista. *Lecture notes in computer science*, pages 359–372.

Gérard Huet. 2005. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *Journal of Functional Programming*, 15(4):573–614.

Mans Hulden. 2009a. *Finite-state machine construction methods and algorithms for phonology and morphology*. Ph.D. thesis.

Mans Hulden. 2009b. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, pages 29–32. Association for Computational Linguistics.

Mans Hulden. 2011. Constraint grammar parsing with left and right sequential finite transducers. In *Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing*, pages 39–47. Association for Computational Linguistics.

Fred Karlsson. 1990. Constraint grammar as a framework for parsing running text. In *Proceedings of the 13th conference on Computational linguistics-Volume 3*, pages 168–173. Association for Computational Linguistics.

George Anton Kiraz. 2001. Compressed storage of sparse finite-state transducers. In *Automata Implementation*, pages 109–121. Springer.

Janne Peltonen. 2011. A finite state constraint grammar parser. In *Proceedings of the NODALIDA 2011 Workshop: Constraint Grammar Applications*, pages 35–40.

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

Pasi Tapanainen. 1999. *Parsing in two frameworks: finite-state and functional dependency grammar*. Ph.D. thesis, University of Helsinki, Department of General Linguistics.

Viktor Trón, András Kornai, György Gyepesi, László Németh, Péter Halácsy, and Dániel Varga. 2005. Hunmorph: open source word analysis. In *Proceedings of the Workshop on Software*, pages 77–85. Association for Computational Linguistics.

Francis M Tyers. 2010. Rule-based Breton to French machine translation. In *Proceedings of the 14th Annual Conference of the European Association of Machine Translation, EAMT10*, pages 174–181.

Anssi Yli-Jyrä. 2011. An efficient constraint grammar parser based on inward deterministic automata. In *Proceedings of the NODALIDA 2011 Workshop: Constraint Grammar Applications*, pages 50–60.

[6] `https://google-melange.appspot.com/gsoc/project/details/google/gsoc2013/davidnemeskey/5764017909923840`