Exploring the Expressivity of Constraint Grammar

Pepijn Kokke
University of Edinburgh
pepijn.kokke@ed.ac.uk

Inari Listenmaa
University of Gothenburg
inari.listenmaa@cse.gu.se

Abstract

We believe that for any formalism which has its roots in linguistics, it is a natural question to ask "how expressive is it?" Therefore, in this paper, we begin to address the question of the expressivity of CG. Aside from the obvious theoretical interest, we envision also practical benefits. For instance, we hope that the FSA→CG conversion tool, described in later sections of this paper, could eventually be developed to generate human-readable CG code from regular expressions or a context-free grammar.

1 Introduction

For any formalism with its root in linguistics, it is natural to ask questions such as "How expressive is it?" or "Where does it sit in the Chomsky hierarchy?" (Chomsky, 1956) In this paper, we begin addressing some of these questions for constraint grammar (Karlsson et al., 1995, CG).

Before we can even consider such a question, there is a problem we must solve. CG was never meant to be a grammar in the generative sense. Instead, it is a tool for analysing and disambiguating strings. This, we believe, explains why the question of the expressivity of CG went unasked and unanswered for a long time. It also gives us our first problem: How do we view CGs generatively? We address this in section 2.

2 Generative Constraint Grammar

We view a constraint grammar CG as generating a formal language $\mathscr{L}$ over an alphabet Σ as follows. We encode words $w \in \Sigma^\star$ as a sequence of cohorts, each of which has one of the symbols of w as a reading. A constraint grammar CG rejects a word if, when we pass its encoding through the CG, we get back the cohort `"<REJECT>"`. A constraint grammar CG accepts a word if it does not reject it. We generate the language $\mathscr{L}$ by passing every $w \in \Sigma^\star$ through the CG, and keeping those which are accepted.

As an example, consider the language $a^\star$ over $\Sigma = \{a, b\}$. This language is encoded by the following constraint grammar:

```
LIST A = "a";
LIST B = "b";
SET LETTER = A OR B;
SELECT A;
ADDCOHORT ("<REJECT>")
    BEFORE LETTER
    IF (-1 (>>>) LINK 1* B);
REMCOHORT LETTER
    IF (-1* ("<REJECT>"));
```

We then encode the input words as a series of letter cohorts with readings (e.g. `"<l>" "a"`, `"<l>" "b"`), and run the grammar. For instance, if we wished to know whether either word in $\{aaa, aab\}$ is part of the language $a^\star$, we would run the following queries:

Input	Output
`"<l>" "a"`	`"<l>" "a"`
`"<l>" "a"`	`"<l>" "a"`
`"<l>" "a"`	`"<l>" "a"`
`"<l>" "a"`	`"<REJECT>"`
`"<l>" "a"`	
`"<l>" "b"`	

As CG is a tool meant for disambiguation, we can leverage its power to run both queries at once:

Input	Output
`"<l>" "a"`	`"<l>" "a"`
`"<l>" "a"`	`"<l>" "a"`
`"<l>" "a" "b"`	`"<l>" "a"`

This is a powerful feature, because it allows us disambiguate based on some formal language $\mathscr{L}$ if we can find the CG which generates it. However,

the limitations of this style become apparent when we look at a run of a CG for the language $\{ab, ba\}$:

Input	Output
"<l>" "a" "b"	"<l>" "a" "b"
"<l>" "a" "b"	"<l>" "a" "b"

While the output contains the interpretations *ab* and *ba*, it also includes *aa* and *bb*. Therefore, while this style is useful for disambiguating using CGs based on formal languages, it is too limited to be used in defining the language which a CG generates.

In light of the idea of using CGs based on formal languages for disambiguating, it seems at odds with the philosophy of CG to reject by replacing the entire input with a single "<REJECT>" cohort. CG generally refuses to remove the last possible reading of a cohort, under the philosophy that *some* information is certainly better than none. However, for the definition of CG as a formal language, we need some sort of distinctive output for rejections. Hence, we arrive at *two* distinct ways to run generative CGs: the method in which we input unambiguous strings, and output "<REJECT>", which is used in the definition of CG as a formal language; and the method in which we input ambiguous strings, and simply disambiguate as far as possible.

3 A lower bound for CG

It should be noted that VISL CG-3 (Bick and Didriksen, 2015; Didriksen, 2014) supports commands such as EXTERNAL, which runs an external executable. It should therefore be obvious that the complete set of CG-3 commands, at least theoretically, can generate any recursively enumerable language. For this reason, we restrict ourselves to a subset of the commands permitted by CG.

In this section, we will only use the REMOVE command with sections, in addition to a single use of the ADDCOHORT command to add the special cohort "<REJECT>", and a single use of the REMCOHORT command to clean up afterwards. We show that, using only these commands, CG is capable of generating some context-free and context-sensitive languages, which establishes a lower bound on the expressivity of CG (see Figure 1).

3.1 Example grammar: $a^n b^n$

Below, we briefly describe the CG which generates the language $a^n b^n$. This CG is defined over the alphabet Σ, in addition to a hidden alphabet Σ'. These hidden symbols are meant to serve as a simple form of memory. When we encode our input words, we tag each cohort with *every* symbol in the hidden alphabet[1], e.g. for some symbol $\ell \in \Sigma$ and $\Sigma' = \{h_1, \ldots, h_n\}$ we would create the cohort "<ℓ>" "h$_1$" ... "h$_n$".

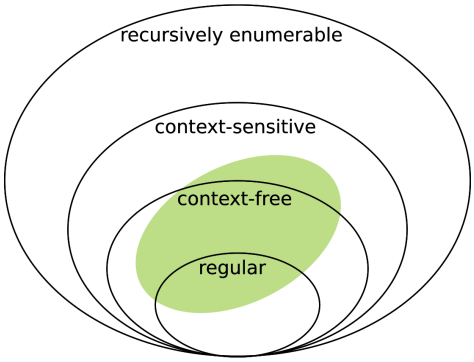

Figure 1: Lower bound on the expressivity of CG.

The CG for $a^n b^n$ uses the hidden alphabet {odd, even, opt_a, opt_b}. These symbols mean that the cohort they are attached to is in an even or odd position, and that *a* or *b* is a legal option for this cohort, respectively. The CG operates as follows:

1. Is the number of characters even? We know the first cohort is odd, and the rest is handled with rules of the form REMOVE even IF (NOT -1 odd). If the last cohort is odd, then discard the sentence. Otherwise continue...

2. The first cohort is certainly *a* and last is certainly *b*, so we can disambiguate the edges: REMOVE opt_b IF (NOT -1 (*)), and REMOVE opt_a IF (NOT 1 (*)).

3. Disambiguate the second cohort as *a* and second-to-last as *b*, the third as *a* and third-to-last as *b*, etc, until the two ends meet in the middle. If every "<a>" is marked with opt_a, and every "<b>" with opt_b, we accept. Otherwise, we reject.

The language $a^n b^n$ is context-free, and therefore CG must at least partly overlap with the context-free languages.

3.2 Example grammar: $a^n b^n c^n$

We can extend the approach used in the previous grammar to write a grammar which accepts

[1] We can automatically add these hidden symbols to our cohorts using a single application of the ADD command.

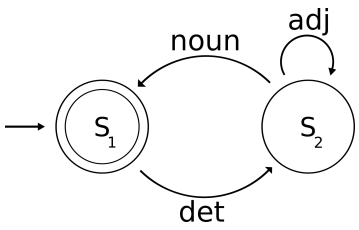

Figure 2: A finite-state automaton describing the regular language `det (adj)* n`.

$a^n b^n c^n$. Essentially, we can adapt the above grammar to find the middle of any input string. Once we have the middle, we can "grow" *a*s from the top and *b*s up from the middle, and *b*s down from the middle and *c*s up from the bottom, until we divide the input into three even chunks. If this ends with all `"<a>"`s marked with `opt_a`, all `"<b>"`s marked with `opt_b`, and all `"<c>"`s marked with `opt_c`, we accept. Otherwise, we reject.

The language $a^n b^n c^n$ is context-sensitive, and therefore CG must at least partly overlap with the context-sensitive languages.

4 Are all regular languages in CG?

In the present section, we propose a method to transform arbitrary finite-state automata into CG. Figure 2 presents an example automaton with $\Sigma = \{det, adj, n\}$, which checks whether some assignment of part-of-speech tags is "valid". We implement a corresponding CG in the following sections.

4.1 Cohorts and sentences

As previously, we define a hidden alphabet $\Sigma' = \{$`opt_det, opt_adj, opt_n`$\}$, and insert the full set Σ' into each cohort as readings. In addition, we introduce *state cohorts*, which contain the full set $S = \{s_1, s_2\}$. For example, the sequence *det n* would be modelled with the following sentence: The rules of the grammar disambiguate both word

`"<s>"`	`"<the>"`	`"<s>"`	`"<book>"`	`"<s>"`
s1	det	s1	n	s1
s2	opt_det	s2	opt_det	s2
	opt_adj		opt_adj	
	opt_n		opt_n	

cohorts and state cohorts. Thus the desired result shows both the accepted string and the path in the automaton, as follows:

`"<s>"`	`"<the>"`	`"<s>"`	`"<book>"`	`"<s>"`
s1	det	s2	n	s1
	opt_det		opt_n	

4.2 Rules

Given that every transition happens between two states, and every state has an incoming and outgoing transition, every rule needs only positions -1 and 1 in its contextual tests. The semantics of the rules are "remove an `opt_POS` tag, if it is *not* surrounded by allowed states", and "remove a state, if it is *not* surrounded by allowed transitions". For the example automaton, the `opt_POS`-rules are as follows:

```
REMOVE...
  opt_det IF (NEGATE -1 S1 LINK 2 S2) ;
  opt_adj IF (NEGATE -1 S2 LINK 2 S2) ;
  opt_n   IF (NEGATE -1 S2 LINK 2 S1) ;
```

The start and end states naturally correspond to the first and last state cohort, and can be trivially disambiguated, in this case both into `s1`. Once we remove a reading from either side of a cohort, some more rules can take action—the context "`s2` on the left side and `s1` on the right side" may be broken by removing either `s2` or `s1`.

If there is only one allowed string of length *n* in the language, then the result should have only one `opt_POS` reading per cohort. The original cohort may be ambiguous or not before adding the hidden readings: in both cases, a mismatch between the remaining opt-reading and the (possibly singleton) list of original readings will be a cause to reject. For instance, {`det`, `opt_det`} accepts the original reading `det`; {`det`, `adj`, `opt_det`} accepts and disambiguates the cohort into `det`; finally, {`adj`, `opt_det`} rejects the whole string as not part of the regular language.

If there are multiple strings of the same length in the language, we have to relax our criteria: if every cohort with a reading `POS` has a corresponding `opt_POS` in the set of remaining readings, such as {`det`, `opt_det`, `opt_adj`}, we accept the string.

4.3 Limitations

As Lager and Nivre (2001) point out, CG has no way of expressing disjunction. Unlike its close cousin FSIG (Koskenniemi, 1990), which would represent a language such as $\{ab, ba\}$ faithfully, CG substitutes uncertainty on the sentence level ("either *ab* or *ba*") with uncertainty in the cohorts: "the first character may be either *a* or *b*, and the

second character may be either *a* or *b*". Given that this is a fundamental design of CG, we do not envision a way out of this limitation, except for the generating multiple CGs per automaton.

However, it is an unclear how much this property would limit us in creating *useful* CG rules out of more expressive grammar formalisms. Applying the rules to real-life ambiguous cohorts is expected to be easier: those cohorts are not $\langle\Sigma\rangle_n$, but some much smaller subset of Σ. Hence, despite the theoretical limitation, we are still hopeful for more applied use cases.

5 Discussion

At the time of writing, the grammars generated by the FSA→CG conversion tool look awkward, and involve a number of extra cohorts and symbols. However, it does give us the ability to quickly generate fragments of CG code which disambiguate or rewrite input based on a regular expression. We are hoping to develop this further, to also include context-free or even mildly context-sensitive grammars. Such CGs could be used as, e.g. as part of a larger constraint grammar. In addition, we would like to focus on making the grammars more human-readable, so that they could be used to quickly generate a basis for a constraint grammar from existing context-free grammars (or equivalent formalisms). This could serve as an alternative to learning grammars from a corpus.

6 Related Work

Tapanainen (1999) gives an account of the expressivity of the contextual tests for 4 different constraint formalisms, including CG. In addition, parsing complexity can be easily defined for a given variant and implementation of CG; see for instance Nemeskey et al. (2014). Yli-Jyrä (2017) presents questions regarding the expressive power of Constraint Grammar, concentrating on the implementation side. To our knowledge, CG as a generative model has not been been approached before.

References

Eckhard Bick and Tino Didriksen. 2015. CG-3 – Beyond Classical Constraint Grammar. In *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015)*.

Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September.

Tino Didriksen, 2014. *Constraint Grammar Manual*. Institute of Language and Communication, University of Southern Denmark.

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.

Kimmo Koskenniemi. 1990. Finite-state parsing and disambiguation. In *Proceedings of 13th International Conference on Computational Linguistics (COLING 1990)*, volume 2, pages 229–232, Stroudsburg, PA, USA. Association for Computational Linguistics.

Torbjörn Lager and Joakim Nivre. 2001. Part of speech tagging from a logical point of view. In *Logical Aspects of Computational Linguistics, 4th International Conference (LACL 2001)*, pages 212–227.

Dávid Márk Nemeskey, Francis Tyers, and Mans Hulden. 2014. Why implementation matters: Evaluation of an open-source constraint grammar parser. In *Proceedings of the 25th International Conference on Computational Linguistics (COLING 2014)*, pages 772–780, Dublin, Ireland, August.

Pasi Tapanainen. 1999. *Parsing in two frameworks: Finite-state and Functional dependency grammar*. Ph.D. thesis, University of Helsinki.

Anssi Yli-Jyrä. 2017. The Power of Constraint Grammars Revisited. In *Proceedings of the Constraint Grammar workshop at the 21th Nordic Conference of Computational Linguistics (NODALIDA 2017)*.