Exploring the Expressivity of Constraint Grammar

Wen Kokke
University of Edinburgh
wen.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. Understanding what CG can and cannot express makes it possible to transform other formalisms to corresponding or approximate CGs, thus making way for new ways of grammar writing, and better reuse of existing language resources.

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
```

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.

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 VISL CG-3 commands, at least theoretically, can generate any recursively enumerable language. For this reason, we will investigate particular subsets of the commands permitted by CG. In sections 3 and 4, we will restrict ourselves to the subset of CG which only uses the REMOVE command with sections, and show this to at least cover all regular languages and some context-free and context-sensitive languages. In section 5, we will restrict ourselves to the subset of CG which only uses the ADDCOHORT and REMCOHORT commands with sections, and show this to be Turing complete.

3 A lower bound for 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).

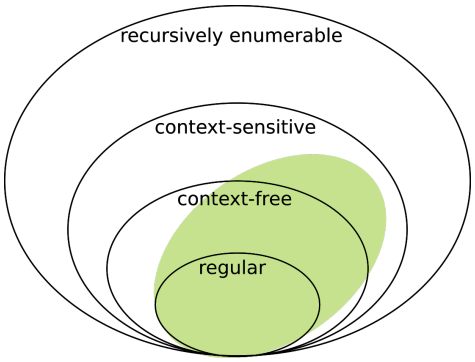

Figure 1: Lower bound on the expressivity of the subset of CG using only REMOVE.

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$_n$".

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.

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

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 $a^n b^n c^n$. 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 any finite-state automata into CG. The translation is implemented in Haskell, and can be found on GitHub[2].

4.1 Finite-state automata

Formally, a finite-state automaton is a 5-tuple

$$\langle \Sigma, S, s_0, \delta, F \rangle.$$

Σ is the alphabet of the automaton, S is a set of states, including a starting state s_0 and a set F of final states. δ is a transition function, which takes one state and one symbol from the alphabet, and returns the state(s) where we can get from the original state with that symbol. The automaton in Figure 2 is presented as follows:

$$\begin{aligned} S &= \{\texttt{s1}, \texttt{s2}\} & \Sigma &= \{\textit{det, adj, n}\} \\ s_0 &= \texttt{s1} & \delta &= \{\texttt{s1} \xrightarrow{\textit{det}} \{\texttt{s2}\}, \\ F &= \{\texttt{s1}\} & & \quad \texttt{s2} \xrightarrow{\textit{adj}} \{\texttt{s2}\}, \\ & & & \quad \texttt{s2} \xrightarrow{\textit{noun}} \{\texttt{s1}\}\} \end{aligned}$$

Informally, the automaton describes a simple set of possible noun phrases: there must be one determiner, one noun, and 0 or more adjectives in between. We implement a corresponding CG in the following sections.

[2] See `https://github.com/inariksit/cgexp`

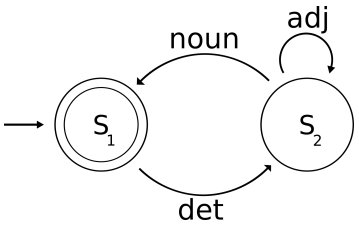

Figure 2: A finite-state automaton describing the regular language `det (adj)* noun`.

4.2 Cohorts and sentences

We encode our input as a sequence of *state cohorts* and *transition cohorts*. Initially, a state cohort contains the full set $S = \{\texttt{s1}, \texttt{s2}\}$ as its readings, and a transition cohort contains the alphabet $\Sigma = \{\textit{det, adj, noun}\}$, or some subset of it. As an example, we generate all 2-letter words recognised by the automaton in Figure 2. The initial maximally ambiguous input for length 2 looks as follows:

```
"<s>" "<w>" "<s>" "<w>" "<s>"
  s1    det   s1    det   s1
  s2    adj   s2    adj   s2
        noun        noun
```

The grammar disambiguates both transition cohorts and state cohorts. Thus the desired result shows both the accepted sequence(s)—*det noun* in this case—and their path(s) in the automaton.

```
"<s>" "<w>" "<s>" "<w>" "<s>"
  s1    det   s2   noun   s1
```

We can easily adapt the disambiguation scheme for real-world ambiguities, such as "the present". The state cohorts are identical, but the transition cohorts contain now some actual word form, and the initial ambiguity is not over the whole Σ, but some subset of it.

```
"<s>" "<the>" "<s>" "<present>" "<s>"
  s1    det     s1      adj       s1
  s2            s2      noun      s2
```

The disambiguation process goes exactly like in the first version, with full Σ in the transition cohorts. Depending on how much the initial input contains ambiguity, the result may be the same, or more disambiguated. For our example, the output is identical.

```
"<s>" "<the>" "<s>" "<present>" "<s>"
  s1    det     s2      noun      s1
```

4.3 Rules

Given that every transition happens between two states, and every state has an incoming and out-

going transition, every rule needs only positions -1 and 1 in its contextual tests. The semantics of the rules are "remove a transition, 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 rules are as follows:

```
REMOVE Det          # Transition rules
    IF (NEGATE -1 S1 LINK 2 S2) ;
REMOVE Adj
    IF (NEGATE -1 S2 LINK 2 S2) ;
REMOVE Noun
    IF (NEGATE -1 S2 LINK 2 S1) ;

REMOVE S1                # State rules
    IF (NEGATE -1 >>> OR Noun
           LINK 2 Det) ;
REMOVE S2
    IF (NEGATE -1 Det OR Adj
           LINK 2 Adj OR Noun) ;
```

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`. One by one, these rules disambiguate the input, removing impossible states and transitions from the cohorts.

4.4 Result

For the final result of the disambiguation, we consider three options: the cohorts may contain the whole alphabet, a well-formed subset or a malformed subset.

Full Σ If there is only one allowed word of length n in the language, then the result will contain only fully disambiguated transition cohorts. Furthermore, if there is only path in the automaton that leads to this word, then also the state cohorts are fully disambiguated.

If there are multiple words of the same length in the language, then we have to relax our criteria: every transition cohort and state cohort in the result may contain multiple readings, but all of them must contribute to some valid word of length n, and its path in the automaton.

Well-formed subset of Σ With well-formed subset, we mean that each cohort contains at least one of the correct readings: {`det`} for "the", and {`adj`,`noun`} for "present". If the initial input is well-formed, then the result will be correct, and may even be disambiguated further than with the full Σ in the transition cohorts.

Malformed subset of Σ Malformed subset has at least one cohort without any correct readings, for example, "the" is missing a `det` reading. This will lead to arbitrary disambiguations, which do not correspond to the automaton. Without a `det` reading in "the", the rule which removes `s2` would trigger in the middle state, leaving us with three `s1` states. `s1-s1-s1` is an impossible path in the automaton, so it would trigger all of the transition rules, and stop only when there is one, arbitrary, reading left in the transition cohorts.

5 Turing Machines in CG?

In the previous sections, we have assumed that CG refers to the subset of VISL CG-3 which uses only the `REMOVE` command. In this section, we will take CG to refer to the subset of VISL CG-3 which uses only the `ADDCOHORT` and `REMCOHORT` commands, and show that this subset is Turing complete. We will do this by implementing a procedure which translates arbitrary Turing machines to CG, taking VISL CG-3 itself as sufficient evidence of the fact that Turing machines can simulate constraint grammars.

The translation we present in this section has been implemented in Haskell, and can be found on GitHub[3]

5.1 A sample Turing machine

We will discuss our translation by means of an example Turing machine. Before we delve into this, however, we will briefly remind the reader of the definition of a Turing machine. A Turing machine is a 7-tuple

$$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle.$$

Q is a finite, non-empty set of states, with a designated starting state $q_0 \in Q$, and a subset $F \subseteq Q$ of accepting states. Γ is a set of tape symbols, with a designated blank symbol b and a subset $\Sigma \subseteq \Gamma \setminus \{b\}$ of input symbols. Lastly, δ is a transition function of the type

$$(Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{\text{Left}, \text{Right}\}.$$

For the remainder of this section, we will use the Turing machine which computes the successors of

[3] See `https://github.com/wenkokke/cgtm`.

binary numbers as an example. This machine is given as follows:

$$Q = \{\texttt{S0}, \texttt{S1}, \texttt{S2}, \text{Halt}\} \quad \Sigma = \{0, 1\}$$
$$\Gamma = \{\_, 0, 1\} \quad q_0 = \texttt{S0}$$
$$b = \_ \quad F = \{\text{Halt}\}$$

The transition function δ is described in table 1. What do these various states do? `S0` and `S2` both move the head of the Turing machine to the start of the number. This leaves `S1` for the actual computation. While in state `S1`, the head will move rightwards, overwriting any `1` it encounters with a `0`, until it reaches either a `0` or the end of the number. It then overwrites this final symbol with a `1`. Table 2 shows the execution trace of our sample Turing machine for the input `1101`, writing the current state *before* the current position of the head.

5.2 Representing the tape and state

We will represent the tape of the Turing machine using the sequence of cell cohorts (written `"<c>"`):

```
"<c>" "<c>" "<c>" "<c>"
 "1"   "1"   "0"   "1"
```

We will store the current state in a special cohort (written `"<s>"`) which we insert right before the cell the Turing machine is currently reading. This means that, e.g. the middle row in table 2 is represented by the following cohorts:

```
"<c>" "<c>" "<c>" "<c>" "<s>" "<c>" "<c>"
 "_"   "_"   "0"   "0"  "S1"  "0"   "1"
```

5.3 Simulating the Turing machine

We start the Turing machine by inserting a cohort with the starting state at the beginning of our input. The starting state for our sample machine is `S0`, so we add the following code to our CG:

```
BEFORE-SECTIONS
ADDCOHORT ("<s>" "S0")
   BEFORE ("<c>") IF (-1 (>>>));
```

Now for the main portion of the Turing machine—simulating the transition function. Since this function is applied iteratively, we will wrap our code in a `SECTION`. We need some way to simulate an infinite tape. Therefore, the first thing we do in each section is check if the current head is near the edge of the tape. If it is, we simply add a new, blank cell:

```
ADDCOHORT ("<c>" "_")
   BEFORE ("<s>")
      IF (-1 (>>>));
ADDCOHORT ("<c>" "_")
    AFTER ("<c>")
      IF (0 (<<<) LINK -1 ("<s>"));
```

We also need some way to distinguish input from output, so before we apply our transition rules, we mark the old state and the old input symbol with the tag `"OLD"`:

```
ADD ("<s>" "OLD") ("<s>");
ADD ("<c>" "OLD") ("<c>")
 IF (-1 ("<s>" "OLD"));
```

We are using an `ADD` command here for clarity, though it is possible to encode this usage of `ADD` using `ADDCOHORT` by simply inserting a specialized cohort (e.g. `"<old>"`) after the cohort we wish to mark, and adjusting all indices and ranges accordingly.

Next, we encode our transition rules. We will translate every entry in our transition function to a pair of rules. The first of these inserts the new state, and the second of these inserts a *new* cell, with whatever we wish to write, after the old cell. For instance, the sixth rule in table 1, which says that "if we are in state 1, and we read a 1, then we write a 0, move the tape to the right, and continue in state 1," is compiled to the following two rules:

```
ADDCOHORT ("<s>" "S1")
   BEFORE ("<c>")
      IF (-2 ("<s>" "S1" "OLD") LINK
           1 ("<c>" "1" "OLD"));
ADDCOHORT ("<c>" "0")
    AFTER ("<c>" "1" "OLD")
      IF (-1 ("<s>" "S1" "OLD"));
```

Note that the first of these rules is in effect responsible for moving the head over the tape. Because of this, a rule for left movement will look slightly different. For instance, the rule which says that "if we are in state 1, and we read a blank, then we write a 1, move the tape to the left, and change to state 2" is compiled to the following two rules:

```
ADDCOHORT ("<s>" "S2")
   BEFORE ("<c>")
      IF (1 ("<s>" "S1" "OLD") LINK
          1 ("<c>" "_" "OLD"));
ADDCOHORT ("<c>" "1")
    AFTER ("<c>" "_" "OLD")
      IF (-1 ("<s>" "S1" "OLD"));
```

We run such a pair of rules for each element in the transition function, and then we finish the SECTION by removing the old state and input cell:

```
REMCOHORT ("<s>" "OLD");
REMCOHORT ("<c>" "OLD");
```

If we wish to know where the head of the machine was located when the program terminated, we can alter these lines to remove any state *except* for the halting states. However, in this instance, we will opt instead to truncate the tape after the execution finishes, by removing any leading or trailing blank cells:

```
AFTER-SECTIONS
REMCOHORT ("<c>" "_") IF (NOT -1* SYM);
REMCOHORT ("<c>" "_") IF (NOT  1* SYM);
```

6 Linear-Bounded Automata in CG?

Linear-bounded automata (LBA) are important, because they accept exactly the class of context-sensitive languages. They are defined as Turing machines whose tape is restricted to the portion containing the input. It therefore seems obvious that we can simulate an LBA by removing the two rules which expand the tape from the transformation outlined in section 5. However, this is not incredibly interesting, as we already know the subset of CG using only ADDCOHORT and REMCOHORT is Turing complete. In this section, we will discuss a different subset of CG which we believe to be sufficiently expressive to cover all context-sensitive grammars. This is the subsets using only ADD and REPLACE.

6.1 LBAs using ADD and REPLACE

In the encoding for Turing machines in section 5 we use ADDCOHORT, as it is the most obvious way to simulate an infinite tape. However, for LBAs, we no longer need an infinite tape. We can require the machine to do all its work with the limited number of cohorts it has been given as its input. This means we can do all the computation by adding and removing tags. We can retain much of the structure we set up for simulating Turing machines:

1. we start by marking the cohort we are currently reading—i.e. the only cohort with a state tag—with "OLD"; then
2. we ADD the next state tag to the cohort to which we are moving; then
3. we REPLACE all tags on the cohort which we left with the output symbol.

And we repeat the above steps until we reach a halting state. This way, we can implement any linear-bounded automaton as a constraint grammar using only ADD and REPLACE.

We can take this idea one step further by replacing any usage of ADD with a usage of REPLACE. We can do this, because LBAs use a finite set of states and a finite alphabet. For instance, we can mark the cohort we are currently reading as "OLD" using a series of REPLACE rules,

$$\forall q \in Q, \forall a \in \Gamma,$$
$$\texttt{REPLACE ("<c>" "}q\texttt{" "}a\texttt{" "OLD")}$$
$$\texttt{("<c>" "}q\texttt{" "}a\texttt{");}$$

Similarly for tagging the next state. However, this does result in a huge blowup in the number of rules, as instead of writing a single rule for each of these uses of ADD, we now write $|Q| \cdot |\Gamma|$ rules, to test every single combination of state and symbol.

Note that we can set up a similar construction using only the commands APPEND and REMOVE, by using readings instead of tags.

7 Discussion

We have shown several different constructions, using different subsets of CG. The resulting grammars are not very readable: they include extra cohorts and symbols, and the logic is spread across rules in a rather obscure way—in contrast to a human-written grammar, where each rule is a self-contained piece of truth about a language. Therefore we do not envision the generated grammars being used as is, but rather as compilation targets. Such CGs could be used as a part of a larger constraint grammar: some sections can be written manually, and others derived from existing grammars. This could serve as an alternative to learning grammars from a corpus. So far we only have a working conversion tool for finite-state automata, but we are hoping to develop this further, to also include context-free or even mildly context-sensitive grammars.

Another question is, even if we had a working conversion system for CFGs, would the result be correct? 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*". If we use such a CG to generate, by feeding it maximally ambiguous cohorts, the result will be overly permissive. We acknowledge that this is a limitation in the expressive power: many languages can only be approximated by CG, not reproduced exactly. Nevertheless, this limitation may not matter so much when disambiguating real-world text, because the cohorts are initially less ambiguous, and leaving genuine ambiguity intact is desired behaviour for CG.

8 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) relates CG to early formal language theory, and provides an independent proof of non-monotonic[4] CG being Turing-complete.

[4] A monotonic variant of CG may only remove readings from cohorts, whereas a non-monotonic variant may add readings or cohorts.

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)*.

State In	Symbol In	Symbol Out	State Out	Move
	Read "_"	Write "_"	"S1"	Right
"S0"	Read "0"	Write "0"	"S0"	Left
	Read "1"	Write "1"	"S0"	Left
	Read "_"	Write "1"	"S2"	Left
"S1"	Read "0"	Write "1"	"S2"	Left
	Read "1"	Write "0"	"S1"	Right
	Read "_"	Write "_"	Halt	Right
"S2"	Read "0"	Write "0"	"S2"	Left
	Read "1"	Write "1"	"S2"	Left

Table 1: Sample Turing machine (binary successor function)

"<c>"	"<s>"	"<c>"	"<s>"	"<c>"	"<s>"	"<c>"	"<s>"	"<c>"	"<s>"	"<c>"
"_"		"_"	"S0"	"1"		"1"		"0"		"1"
"_"	"S0"	"_"		"1"		"1"		"0"		"1"
"_"		"_"	"S1"	"1"		"1"		"0"		"1"
"_"		"_"		"0"	"S1"	"1"		"0"		"1"
"_"		"_"		"0"		"0"	"S1"	"0"		"1"
"_"		"_"		"0"	"S2"	"0"		"1"		"1"
"_"		"_"	"S2"	"0"		"0"		"1"		"1"
"_"	"S2"	"_"		"0"		"0"		"1"		"1"
"_"		"_"	"S2"	"0"		"0"		"1"		"1"

Table 2: Execution trace of a Turing machine (see table 1) for input 1101