Constraint Grammar parsing with left and right sequential finite transducers

Mans Hulden
University of Helsinki
mans.hulden@helsinki.fi

Abstract

We propose an approach to parsing Constraint Grammars using finite-state transducers and report on a compiler that converts Constraint Grammar rules into transducer representations. The resulting transducers are further optimized by conversion to left and right sequential transducers. Using the method, we show that we can improve on the worst-case asymptotic bound of Constraint Grammar parsing from cubic to quadratic in the length of input sentences.

1 Introduction

The Constraint Grammar (CG) paradigm (Karlsson, 1990) is a popular formalism for performing part-of-speech disambiguation, surface syntactic tagging, and certain forms of dependency analysis. A CG is a collection of hand-written disambiguation rules for part-of-speech or syntactic functions. The popularity of CGs is explained by a few factors. They typically achieve quite high F-measures on unrestricted text, especially for free word-order languages (Chanod and Tapanainen, 1995; Samuelsson and Voutilainen, 1997). Constraint Grammars can also be developed by linguists rather quickly, even for languages that have only meager resources available as regards tagged or parsed corpora, although it is hard to come by exact measures of how much effort development requires. One drawback to using CG, however, is that applying one to disambiguate input text tends to be very slow: for example, the Apertium project (Forcada et al., 2009), which offers the option of using both n-gram models and CG (by way of the *vislcg3* compiler (Bick, 2000)), reports that using n-gram models currently results in ten times faster operation, although at the cost of a loss in accuracy.

In this paper, we describe a process of compiling individual CG rules into finite-state transducers (FSTs) that perform the corresponding disambiguation task on an ambiguous input sentence. Using this approach, we can improve the worst-case running time of a CG parser to quadratic in the length of a sentence, down from the cubic time requirement reported earlier (Tapanainen, 1999). The method presented here implements faithfully all the operations allowed in the CG-2 system documented in Tapanainen (1996). The same approach can be used for various extensions and variants of the Constraint Grammar paradigm.

The idea of representing CG rules as FSTs has been suggested before (Karttunen, 1998), but to our knowledge this implementation represents the first time the idea has been tried in practice.[1] We also show that after compiling a collection of CG rules into their equivalent FSTs, the individual transducers can further be converted into left and right sequential transducers which greatly improves the speed of application of a rule.

In the following, we give a brief overview of the CG formalism, discuss previous work and CG parsers, provide an account of our method, and finally report on some practical experiments in compiling large-scale grammars into FSTs with our CG-rule-to-transducer compiler.

2 Constraint Grammar parsers

A Constraint Grammar parser occupies the central role of a system in the CG framework. A CG system

[1] However, Peltonen (2011) has recently implemented a subset of CG-2 as FSTs using a different method.

Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing, pages 39–47,
Blois (France), July 12-15, 2011.

is usually intended to produce part-of-speech tagging and surface syntactic tagging from unrestricted text. Generally, the text to be processed is first tokenized and subjected to morphological analysis, possibly by external tools, producing an output where words are marked with ambiguous, alternative readings. This output is then passed as input to a CG parser component. Figure 1 (left) shows an example of intended input to a CG parser where each indented line following a lemma represents an alternative morphological and surface syntactic reading of that lemma; an entire group of alternative readings, such as the five readings for the word *people* in the figure is called a *cohort*. Figure 1 (right) shows the desired output of a CG disambiguator: each cohort has been reduced to contain only one reading.

2.1 Constraint grammar rules

A CG parser operates by removing readings, or by selecting readings (removing the others) according to a set of CG rules. In its standard form there exists only these two types of rules (`SELECT` and `REMOVE`). How the rules operate is further conditioned by constraints that dictate in which environment a rule is triggered. A simple CG rule such as:

```
REMOVE (V) IF (NOT *-1 sub-cl-mark)
              (1C (VFIN)) ;
```

would remove all readings that contain the tag `V`, if there (a) is no subordinate clause mark anywhere to the left (indicated by the rule scope (NOT $*-1$), *and* (b) the next cohort to the right contains the tag VFIN in *all* its readings (signaled by 1C (VFIN)). Such a rule would, for instance, disambiguate the word *people* in the example sentence in Figure 1, removing all other readings except the noun reading. Rules can also refer to the word-forms or the lemmas in their environments. Traditionally, the word-forms are quoted while the lemmas are enclosed in brackets and quotation marks (as in ``<counselors>'' vs. ``counselor'' in fig. 1).

In the example above, only morphological tags are being used, but the same formalism of constraints is often used to disambiguate additional, syntactically motivated tags as well, including tags that mark phrases and dependencies (Tapanainen, 1999; Bick, 2000). Additional features in the rule formalism include `LINK` contexts, Boolean operations, and `BARRIER` specifications. For example, a more complete rule such as:

```
"<word>" REMOVE (X) IF
(*1 A BARRIER C LINK *1 B BARRIER C);
```

would remove the tag `X` for the word-form `word` if the first tag `A` were followed by a `B` somewhere to the right, and there was no `C` before the `B`, except if the first `A`-tagged reading also contained `C`.

It is also possible to add and modify tags to cohorts using `ADD` and `MAP` operations, which work exactly as the `SELECT` and `REMOVE` operations as regards the contextual target specification.

2.2 Parser operation

Given a collection of CG rules, the job of the parser is to apply each rule to the set of input cohorts representing an ambiguous sentence as in Figure 1, and remove or select readings as the rule dictates. The formalism specifies no particular rule ordering per se, and different implementations of the CG formalism apply rules in varying orders (Bick, 2000). In this respect, it is up to the grammar writer to design the rules so that they operate correctly no matter in what order they are called upon. The parser iterates rule application and removes readings until no rule can perform any further disambiguation, or until each cohort contains only one reading. Naturally, since no rule order is explicit, most parser implementations (Tapanainen, 1996; Bick, 2000) tend to use complex techniques to predict if a certain rule can apply at all to avoid the costly process of checking each reading and its respective contexts in an input sentence against a rule for possible removal or selection.

2.3 Computational complexity

Tapanainen (1999) gives the following complexity analysis for his CG-2 parsing system. Assume that a sentence of length n contains maximally k different readings of a token, and is to be disambiguated by a grammar consisting of G rules. Then, testing whether to keep or discard a reading with respect to a single rule can be done in $O(nk)$, with respect to all rules, in $O(Gnk)$, and with respect to all rules and all tokens in $O(n^2Gk)$. Now, in the worst case, applying all rules to all alternative readings only results in the discarding of a single reading. Hence, the process must in some cases be repeated $n(k-1)$ times,

```
"<Business>"
        "business" <*> N NOM SG
"<people>"
        "people" N NOM SG/PL
        "people" V PRES -SG3 VFIN
        "people" V IMP VFIN
        "people" V SUBJUNCTIVE VFIN
        "people" V INF
"<can>"
        "can" V AUXMOD Pres VFIN

"<play>"
        "play" N NOM SG
        "play" V PRES -SG3 VFIN
        "play" V IMP VFIN
        "play" V SUBJUNCTIVE VFIN
        "play" V INF
"<a>"
        "a" <Indef> DET CENTRAL ART SG
"<role>"
        "role" <Count> N NOM SG
"<as>"
        "as" ADV AD-A>
        "as" <**CLB> CS
        "as" PREP
"<counselors>"
        "counselor" <DER:or> <Count> N NOM PL
"<and>"
        "and" CC
"<teachers>"
        "teacher" <DER:er> <Count> N NOM PL
"<.>"
        "." PUNCT Pun
```

```
"<Business>"
        "business" <*>  N NOM SG
"<people>"
        "people" N NOM SG/PL
"<can>"
        "can" V AUXMOD Pres VFIN
"<play>"
        "play" V INF
"<a>"
        "a" <Indef> DET CENTRAL ART SG
"<role>"
        "role" <Count> N NOM SG
"<as>"
        "as" PREP
"<counselors>"
        "counselor" <DER:or> <Count> N NOM PL
"<and>"
        "and" CC
"<teachers>"
        "teacher" <DER:er> <Count> N NOM PL
"<.>"
        "." PUNCT Pun
```

Figure 1: Example input (left) and output (right) from a Constraint Grammar disambiguator.

yielding a total complexity of $O(n^3Gk^2)$. As mentioned above there are various heuristics one can use to avoid blindly testing rules against readings where they cannot apply, but none that guarantee a lower complexity.

3 Related work

Many constraint-based tagging systems can be speeded up by appropriate use of finite-state transducers. For example, Roche and Schabes (1995) show that a Brill tagger (Brill, 1995) can be applied in linear time by constructing a sequential (input-deterministic) transducer that performs the same task as applying a set of transformation-based learning (TBL) rules that change tags according to contextual specifications. This method does not, however, transfer to the problem of CG implementations: for one, TBL rules are vastly simpler in their expressive power, limited only to a few simple templatic statements of tag replacement, while the CG formalism allows for an unlimited number of Boolean and linking constraints; secondly, TBL rules target tags, not words, while CG allows for rules to target any mix of both; thirdly, TBL rules only replace single tags with other single tags and do not remove tags from sets of alternative tags.[2]

Additionally, Koskenniemi (1990); Koskenniemi et al. (1992) have proposed a constraint-based method for surface-syntactic tagging that can be directly implemented—at least in theory—as the intersection of constraints encoded by finite automata. This formalism has been called alternatively by the name *finite-state intersection grammar* and *parallel constraint grammar*, and has later been pursued

[2]This last circumstance is actually only a theoretical inequivalence: a set of CG tags could conceivably be encoded as a single symbol, and the problem of removing tags from a set of tags could be reduced to changing set-representing tags into other such tags, bringing TBL closer to the CG formalism. However, then each possible word targeted (since in CG, words are considered tags as well) would have to be a member of this powerset of tags, causing an exponential explosion in the tag alphabet size.

by Tapanainen (1997) and Yli-Jyrä (2005), among others. While a finite-state implementation of this formalism in theory also offers linear-time performance, it remains unclear whether the massive constants stemming from an explosion in the size of the automata that encode intermediate results can be avoided and a practical parsing method produced (Yli-Jyrä, 2005).

4 Overview of method

Previous CG compilers operate by choosing a rule and a reading and scanning the context to the left and the right to decide if the reading should be removed, possibly using additional information in the form of bookkeeping of where and which rules can potentially apply in a given sentence. In contrast, the approach taken here is to construct a FST from each rule. This transducer is designed so that it acts upon a complete input string representing a sentence and ambiguous cohorts. In one go, it applies the rule to all the readings of the sentence, removing the readings the rule dictates and retaining all others.

For reasons of simplicity, instead of directly operating on the types of inputs given in Figure 1, we assume that the transducer will act upon a slightly modified, more compact string representation of an ambiguous sentence. Here, an entire sentence is represented as a single-line string, with certain delimiter marks for separating cohorts and lemmas. The changes can be illustrated in the following snippet: the cohort

```
"<as>"
      "as" ADV
      "as" PREP
```

is represented as a string in the format

```
$0$ "<as>" #BOC# |
#0# "as" ADV     |
#0# "as" PREP    | #EOC#
```

That is, we have symbols for representing beginnings and endings of cohorts (`#BOC#`, `#EOC#`), and delimiters between every reading (`|`). Additionally, the symbol `#X#` is used to mark readings that have been removed, and the symbol `#0#` readings that are still possible. The choice of symbols is arbitrary; their role is only to make the data representation compact and suitable for targeting by regular language/FSTs.

The task of each constructed rule transducer, then, is actually only to change `#0#`-symbols into `#X#`-symbols for those readings that should be removed by the rule, of course making sure we never remove the last remaining reading as per CG requirements.

For example, removal of the `ADV`-reading for the word *as* in the above cohort would result in the output string:

```
$0$ "<as>" #BOC#   |
#X# "as"   ADV     |
#0# "as"   PREP    | #EOC#
```

5 Left/right sequential transducers

The output of the compilation process is a transducer that removes readings as the corresponding rule requires—in practice only changing `#0#` symbols into `#X#` symbols wherever warranted. However, if the rule contexts are complicated, this transducer may contain many alternative paths with `#0#:#0#` or `#0#:#X#` labels going out from the same state, only one of which contains a path to a final state with any input string: i.e. the transducer is not sequential. This is because more context to the right needs to be seen by the transducer to decide whether to retain or remove a reading. In the case of large rules, this may involve substantial backtracking when applying a transducer against an input string. The time taken to apply a rule transducer is still linear in the length of the string, but may hide a large constant, which is in effect the size of the transducer.

However, we a priori know that each rule transducer is *functional*, i.e. that each input string maps to maximally one output string (rules are never ambiguous in their action). Such transducers T can be broken up by a process called bimachine factorization into two unambiguous transducers: a left sequential T_l and a right sequential one T_r, such that the effect of the original transducer is produced by first applying the input word to T_l, and then applying T_r to the reverse of T_l's output (Schützenberger, 1961; Reutenauer and Schützenberger, 1991; Roche, 1995; Kempe, 2001). In other words, the two separate transducers fulfill the condition that:

$$T = T_l \circ \mathit{Reverse}(T_r) \quad (1)$$

Performing this factorization of the rule transducers then allows further efficiency gains. For instance,

as table 1 shows, some rules with long contexts produce transducers with thousands of states. Converting these rules to the equivalent left and right sequential ones removes a large time constant from the cost of applying a rule. It could be noted that the rule transducers are also directly *sequentiable* into a single sequential transducer (Schützenberger, 1977), and we could apply such a sequentialization algorithm (Mohri, 1997) on them as well. Sequentializing an FST in effect postpones ambiguous output along transducer paths until the ambiguity is resolved, emitting zeroes during that time. Performing this type of sequentialization on rule FSTs is in practice impossible, however. As each ambiguity may last several cohorts ahead, the equivalent sequential transducer must "remember" arbitrary non-outputted strings for a long time, and will be exponential in size to the original one. By contrast, the resulting left and right sequential rule FSTs are actually smaller than the original rule FSTs.

6 Construction

Since each rule can operate in complex ways, we break down the process of compiling a rule into several smaller transducers which are joined by composition (∘). This is similar to techniques used for compiling phonological rewrite rules into FSTs (Kaplan and Kay, 1994; Kempe and Karttunen, 1996). The entire construction process can be encapsulated in the composition of a few auxiliary transducers. Compilation of a basic rule of the format

```
SELECT/REMOVE (X) IF
(SCOPE#1 COND#1) ... (SCOPE#n COND#n)
```

can be expressed with the general construction

$$\begin{aligned} &\texttt{MarkFormTarget} \circ \\ &\qquad \texttt{Constrain} \circ \\ &\texttt{Cond}_1 \circ \ldots \circ \texttt{Cond}_n \end{aligned} \tag{2}$$

These operate as follows:

- `MarkFormTarget` is a transducer that changes `#0#`-symbols temporarily to `#1#`-symbols (signaling pending removal) for those cohorts that contain the target reading (if the rule is a REMOVE rule), or for retention (if it is a SELECT rule).
- `Constrain` changes `#1#`-symbols back into `#0#` symbols whenever the last reading would be removed.
- $\texttt{Condition}_k$ changes the corresponding temporary symbols into `#X#`-symbols whenever all the conditions are met for removal, otherwise changing them back to `#0#`-symbols.

The actual conditions expressed in the `Cond` transducer are fairly straightforward to express as Boolean combinations of regular languages since we have explicit symbols marking the beginnings and endings of cohorts as well as readings. Each condition for a rule firing—e.g. something occurring n cohorts (or more) to the left/right—can then be expressed in terms of the auxiliary symbols that separate cohorts and readings.

6.1 Detailed example

Figure 2 contains a working example of the rule compilation strategy in the form of a script in the Xerox formalism compilable with either the *xfst* (Beesley and Karttunen, 2003) or *foma* (Hulden, 2009) FST toolkits. The majority of the example consists of function and transducer definitions common for compiling any CG-rule, and the last few lines exemplify the actual compilation of the rules. Briefly, compiling a rule with the example code, entails as a preliminary the composition of the following transducers:

- `InitialFilter` disallows, for efficiency reasons, all input that is not correctly formatted.
- `MarkFormTarget(Wordform,Target)` is a function that performs the provisional marking of all target readings and cohorts that could be affected by the rule, given the wordform and the target tag.
- `ConstrainS` for `SELECT`-rules (and `ConstrainR` for `REMOVE`-rules), is a transducer that checks that we would not remove the last reading from any cohort, should the rule be successful and reverts auxiliaries `#1#` to `#0#` in this case. Also, each potentially affected cohort which is by default headed by `$0$`, is mapped ambiguously to `$A$` or

`$R$`. These symbols serve to denote whether a change in that cohort is to be later accepted or rejected, based on the rule contexts.

These three transducers are again composed with a transducer that restricts the occurrence of the `$A$`-symbol to those cohorts where the rule contexts are in place. This is followed by the composition with a `Cleanup`-transducer that restores all auxiliaries, leaving all readings to only be marked `#0#` (current) or `#X#` (removed), and all heads marked `$0$`.

The actual implementation is a stand-alone compiler written in C using *flex* and the *foma* API for the transducer construction functions. It handles additional chained `LINK` contexts, supports set definitions as is the case in the standard CG variants. These additions require dynamic insertions of auxiliary symbols based on the number of linked contexts and defined sets and cannot be captured in static scripts. However, the compilation method and the use of auxiliary symbols is identical in the example script in figure 2. The outputs of the example script are non-sequential transducers that model CG rules that can later be converted to left and right sequential ones for faster application speed.

7 Analysis

Assuming a grammar with G rules and a maximum of k possible readings per word, applying one rule transducer to an entire sentence of n possibly k-way ambiguous words takes time $O(nk)$: we apply the transducer (or the left-sequential and right-sequential transducers) to the string representing the sentence whose string representation length is maximally nk in linear time. Now, applying an entire set of G rules in some order can be done in $O(Gnk)$ time. Making no assumptions about the structure of the input, in the worst case one such round of disambiguation only removes a single reading from a single cohort. Applying the entire set of disambiguation rules must then be done (in the worst case) $n(k-1)$ times. Hence, the total time required to be guaranteed of disambiguation of a sentence is of the order $O(Gn^2k^2)$.

The improvement over the prior parsers that operate in $O(Gn^3k^2)$ as analyzed in Tapanainen (1999) comes precisely from the ability to compile a constraint rule into a FST. In that earlier analysis, it was assumed that it takes $O(nk)$ time to resolve whether to keep or discard some chosen alternative reading in a cohort. That is, the underlying idea was to test each *reading* for possible removal separately. The improvement—that we can apply one rule to *all* the cohorts and readings in a sentence in time $O(nk)$—is due to the transducer representation of the rule action.

SC	n		$-n$		$*n$		$*-n$	
	$\|T\|$	$\|B\|$	$\|T\|$	$\|B\|$	$\|T\|$	$\|B\|$	$\|T\|$	$\|B\|$
1	72	44	39	37	47	32	27	31
2	214	77	77	61	74	38	33	37
3	640	143	153	109	108	44	39	43
4	1918	275	305	205	148	50	45	49
5	5752	539	609	397	194	56	51	55
6	17254	1067	1217	781	246	62	57	61
7	51760	2123	2433	1549	304	68	63	67

Table 1: Example sizes (number of states) of single rules of varying left and right scope represented as transducers, both individually and as separate left-sequential and right-sequential transducers. The $|T|$ represents the size of the single transducer, and the $|B|$-columns the sums of the sizes of the LR-sequential ones.

Additionally, the constant G can in theory be eliminated. Given a set of rules represented as transducers, $R_1 \ldots R_n$, these rules can be combined into a larger transducer R by composition:

$$R_1 \circ \ldots \circ R_n \tag{3}$$

Subsequently, this transducer R can be converted into a left and a right sequential transducer as above, yielding $O(n^2k^2)$. In practice, such a construction is not feasible, however, because the composed transducer invariably becomes too large in any actual grammar. The approach is still partially useful as our practical experiments show that rules that target the same tags can be composed without undue growth in the composite transducer. In actual grammars, it is often the case that a large number of different rules operate on removal or selection of the same tag, and in such a case, the individual rule transducers can further be grouped and combined to a certain extent.

```
############################################################
# Auxiliary definitions                                    #
############################################################
# $0$ = heads all cohorts                                  #
# $1$ = temporary auxiliary for marking cohort             #
# $A$ = temporary auxiliary for marking "acceptance" of rule #
# $R$ = temporary auxiliary for marking "rejection" of rule #
# #0# = marks all readings that are alive                  #
# #X# = marks all dead readings                            #
# #1# = temporary auxiliary: marks readings that are about to #
#       be retained                                        #
# #2# = temporary auxiliary: marks readings that are about to #
#       be removed                                         #
# #BOC# = marks a beginning of each cohort                 #
# #EOC# = marks the end of each cohort                     #
############################################################

define DEL "|" ;
define ALIVE ["#0# "|"#1# "|"#2# "] ;
define AUX "$0$ "|"$1$ "|"$A$ "|"$R$ "|
          "#0# "|"#1# "|"#X# "|"#2# "| DEL |"#BOC# "|"#EOC# ";

define InitialFilter  ~$AUX ["$0$ " ~$AUX "#BOC# "
       ([DEL ["#0# "|"#X# "] ~$AUX]+ DEL) "#EOC# "]* ;

define NoMarkedReading ~$["#1# "|"#EOC# "] "#EOC# " ;
define NoLiveReading   ~$["#0# "|"#EOC# "] "#EOC# " ;
define NoMarkedHead    "$0$ " ~$["#EOC# "] ;
define MarkedHead      "$1$ " ~$["#EOC# "] ;

define ConstrainS      "$1$ " -> "$0$ " || _ NoMarkedReading |
                                           NoLiveReading .o.
        "#1# " -> "#0# " || NoMarkedHead _                  .o.
        "#0# " -> "#1# ","#1# " -> "#0# " || MarkedHead _   .o.
        "$1$ " -> ["$A$ "|"$R$ "]                           .o.
        "#1# " -> "#2# " || "$A$ " \"#EOC# "* _ ;

define ConstrainR   "$1$ " -> "$0$ " || _ NoMarkedReading  .o.
                    "#1# " -> "#0# " || NoMarkedHead _     .o.
                    "$1$ " -> "$0$ " || _ NoMarkedReading  .o.
                    "$1$ " -> "$0$ " || _ NoLiveReading    .o.
                    "#1# " -> "#0# " || NoMarkedHead _     .o.
                    "$1$ " -> ["$A$ "|"$R$ "]              .o.
                    "#1# " -> "#2# " || "$A$ " \"#EOC# "* _ ;
```

```
define Cleanup  "#1# " -> "#0# " .o. "#2# " -> "#X# " .o.
                "$A$ "|"$R$ " -> "$0$ ";

define MarkFormTarget(WORDFORM,TARGET)
                           "$0$ " -> "$1$ " || _ WORDFORM .o.
                           "#0# " -> "#1# " || _ TARGET ;
define InvCR [[?* ["$A$ ":"$R$ "] [?|["$A$ ":"$R$ "]]*]]   .o.
             "#2# "->"#1# " || "$R$ " \"#EOC# "* _ ;
define CompileRule(X) X .o. ~[X .o. InvCR].l  .o. Cleanup;
define MATCHCOHORT(X) [\"#BOC# "* "#BOC# " \"#EOC# "* DEL ALIVE
                       \DEL* X \"#EOC# "* "#EOC# " ~$"#BOC# "];
define MATCHCOHORTC(X) [\"#BOC# "* "#BOC# " [[DEL DEAD \DEL*]*
                  [DEL ALIVE \DEL* X \DEL*] [DEL DEAD \DEL*]*]+
                   DEL "#EOC# " ~$"#BOC# "];
define ANYCOHORT [\"#BOC# "* "#BOC# "
                  \"#EOC# "* "#EOC# " ~$"#BOC# "];

########################################################
# Actual compilation of an example rule using the above #
# auxiliary functions and definitions                  #
########################################################

# Rule 1: SELECT (V) IF (1 (ADJ));
define Rule1Pre InitialFilter .o.
                MarkFormTarget(?*, \DEL* "V " \DEL*) .o.
                ConstrainS .o.
       "$A$ " => _ ANYCOHORT MATCHCOHORT(\DEL* "ADJ " \DEL*);
regex  CompileRule(Rule1Pre);

# Rule 2: SELECT (X) IF (*-1C (A) BARRIER (B)) (1 (C));
define Rule21Pre InitialFilter                  .o.
       MarkFormTarget(?*,\DEL* "X " \DEL*)      .o.
       ConstrainS                               .o.
       "$A$ " => MATCHCOHORTC(\DEL* "A " \DEL*)
       [ANYCOHORT - MATCHCOHORT(\DEL* "B " \DEL*)]* _ ;

define Rule22Pre InitialFilter             .o.
       MarkFormTarget(?*,\DEL* "X " \DEL*) .o.
       ConstrainS                          .o.
       "$A$ " => _ ANYCOHORT MATCHCOHORT(\DEL* "C " \DEL*);

# Rule is split into two parts that are intersected
regex CompileRule(Rule21Pre & Rule22Pre);
```

Figure 2: Complete *foma* code example that compiles two different CG rules with the method.

8 Some practical experiments

8.1 Grammar compilation

We have built a CG-to-transducer compiler that converts rules in the CG-2 format (Tapanainen, 1996) into the type of FSTs discussed above. The compiler itself relies on low-level finite-state machine construction functions available in the *foma* FST library (Hulden, 2009). To test the compiler against large-scale grammars, we have run it on Constraint Grammars of Finnish (Karlsson, 1990),[3] and Basque (Aduriz et al., 1997). Both grammars are quite large: the Finnish grammar consists of 1,019 rules, and the Basque of 1,760 rules. Table 2 shows the results of compiling all rules into individual transducers. In the table, what is given is the sum total of states and transitions of all transducers ($\Sigma|T|$), and the sum total of states and transitions for the left and right sequential transducers ($\Sigma|B|$). As can be seen, the sequentialized transducers together are substantially smaller than the non-sequentialized transducers. Compilation time for the respective grammars is currently 2 min 59 sec. (Basque) and 1 min 09 sec. (Finnish).[4]

8.2 Rule transducer size growth

Naturally, there is a limit to the complexity of rules that can be compiled into FSTs. Rules that depend on long-distance left and right contexts will grow quickly in size when represented as FSTs. Table 1 shows the sizes of different transducers compiled from a single rule type

```
SELECT (X) IF (SCOPE (Y))        (4)
```

where `SCOPE` represents various rule scopes. For instance, the scope `*-3` (condition holds three or more words to the left) results in a transducer of 39 states, and a left and right sequential transducer whose sum total of states are 43. Complex rules with

[3] Converted from the original to the more modern notation by Trond Trosterud at the University of Tromsø in 2010.

[4] on a 2.8GHz Intel Core 2 Duo.

	Basque	Finnish
Rules	1,760	1,019
$\Sigma\|T\|$	469,938 states	231,786 states
	11,229,332 trans.	2,741,165 trans.
$\Sigma\|B\|$	213,445 states	102,913 states
	4,812,185 trans.	1,230,118 trans.

Table 2: Sums of sizes of resulting transducers with two large-scale grammars.

multiple conditions may grow larger than the simple, single-context rules in the table, but nevertheless, the results indicate that most grammars should be representable as FSTs in practice. In our test grammars, the longest scope ever used for a condition was 6 cohorts to the right (in Basque)—although e.g. Voutilainen (1999) reports on sometimes needing slightly longer contexts than this for a grammar of English (max. 9).

Since the bimachine factorization used in constructing the left and right sequential transducers introduces unique auxiliary symbols to signal pending ambiguity during the left-to-right pass, which is later resolved, there is some slight growth of the alphabet in these transducers. However, this growth is fairly small: the sequentialization of all the rules in the two grammars tested could be performed with a maximum of 8 auxiliary symbols, usually only one or two for the majority of the rules.

8.3 Grammar analysis

The fact that we can compile each rule in a Constraint Grammar into a finite state transducer also yields other benefits apart from rule application speed. Grammars can be analyzed with respect to errors in detailed fashion with methods that go beyond existing debugging capabilities in CG parsers. For example, the formalism allows for vacuous rules, i.e. rules that never act on any input. Consider, for example: `SELECT (X) IF (NOT 0 (X)).`

Such rules are quite a common redundancy in actual CG grammars and tend to go undetected. While the above rule is easily seen at first glance to be vacuous, more complex rules are more demanding to analyze in this respect. For example, the rule

```
SELECT (ADB) IF (0C ADJ-ADB)
(-1 KASEZGRAM OR ERG OR PAR);
```

was encountered in an actual grammar. It is indeed vacuous, but to detect this, we need to analyze the sets `ADJ-ADB`, `KASEZGRAM`, `ERG`, and `PAR`, as well as the logic of the rule.

Using finite-state techniques, we can calculate, for a rule transducer `R` its intersection with the set of all transducers that change `#0#` symbols into `#X#`-symbols.

$$R \cap (?{:}?^* \texttt{\#0\#:\#X\#} ?{:}?^*) \tag{5}$$

yielding a transducer whose domain contains all the inputs that are affected by the rule `R`, and allowing us to answer the question whether the rule is vacuous. Similar techniques can be used to analyze rule redundancy (are two superficially distinct rules equivalent), and rule subsumption; does R_1 subsume R_2, making it redundant, or do rules R_1 and R_2 together act identically to R_3 alone, and so forth.

9 Conclusion & future work

We have presented a method for compiling individual Constraint Grammar rules into finite-state transducers. This reduces the worst-case time requirements for CG parsing from cubic to quadratic. The possibility of further conversion of rule transducers into left and right sequential ones cuts down on the time constants involved in rule disambiguation. Testing an implementation against wide-coverage grammars seems to indicate that the method is practical even for large grammars. Integrating the approach proposed here with earlier strategies to CG-parsing—most important being efficient tracking of which rules can potentially apply to an input at any given stage to avoid applying transducers unnecessarily—remains an important next practical step.

Acknowledgements

I am grateful to Atro Voutilainen for sharing his insights and providing me with relevant examples and sources, and also to the IXA group at the University of the Basque Country for providing grammars and resources. This research has received funding from the European Commission's 7th Framework Program under grant agreement no. 238405 (CLARA).

References

Aduriz, I., Arriola, J. M., Artola, X., de Ilarraza, A. D., K., G., and M., M. (1997). Morphosyntactic disambiguation for Basque based on the Constraint Grammar Formalism. In *Proceedings of Recent Advances in NLP (RANLP97)*.

Beesley, K. R. and Karttunen, L. (2003). *Finite State Morphology*. CSLI Publications, Stanford, CA.

Bick, E. (2000). *The Parsing System "Palavras": Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework*. Aarhus University Press.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

Chanod, J. P. and Tapanainen, P. (1995). Tagging French: comparing a statistical and a constraint-based method. In *Proceedings of the 7th EACL*, pages 149–156.

Forcada, M. L., Tyers, F. M., and Ramírez-Sánchez, G. (2009). The free/open-source machine translation platform Apertium: Five years on. In *Proceedings of FreeRBMT'09*, pages 3–10.

Hulden, M. (2009). Foma: a finite-state compiler and library. In *Proceedings of the 12th EACL*, pages 29–32.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Karlsson, F. (1990). Constraint grammar as a framework for parsing running text. In *COLING '90*, volume 3, pages 168–173, Helsinki, Finland.

Karttunen, L. (1998). The proper treatment of optimality theory in computational phonology. In *Proceedings of FSMNLP*.

Kempe, A. (2001). Factorization of ambiguous finite-state transducers. *Lecture Notes in Computer Science*, 2088:170–181.

Kempe, A. and Karttunen, L. (1996). Parallel replacement in finite state calculus. In *Proceedings of the 34th ACL*.

Koskenniemi, K. (1990). Finite-state parsing and disambiguation. In *COLING '90*, pages 229–232.

Koskenniemi, K., Tapanainen, P., and Voutilainen, A. (1992). Compiling and using finite-state syntactic rules. In *COLING '92*, pages 156–162.

Mohri, M. (1997). On the use of sequential transducers in natural language processing. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, pages 355–382. MIT Press.

Peltonen, J. (2011). *Rajoitekielioppien toteutuksesta äärellistilaisin menetelmin [On the Implementation of Constraint Grammars Using Finite State Methods]*. MA Thesis, University of Helsinki.

Reutenauer, C. and Schützenberger, M.-P. (1991). Minimization of rational word functions. *SIAM Journal of Computing*, 20(4):669–685.

Roche, E. (1995). Factorization of Finite-State Transducers. *Mitsubishi Electric Research Laboratories*, pages 1–13.

Roche, E. and Schabes, Y. (1995). Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253.

Samuelsson, C. and Voutilainen, A. (1997). Comparing a linguistic and a stochastic tagger. In *Proceedings of the 8th EACL*, pages 246–253.

Schützenberger, M.-P. (1961). A remark on finite transducers. *Information and Control*, 4:185–196.

Schützenberger, M.-P. (1977). Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4:47–57.

Tapanainen, P. (1996). *The Constraint Grammar Parser CG-2*. Publications 27, Department of General Linguistics. University of Helsinki.

Tapanainen, P. (1997). Applying a finite-state intersection grammar. In Roche, E. and Schabes, Y., editors, *Finite-State Language Processing*, pages 311–327. MIT Press.

Tapanainen, P. (1999). *Parsing in two frameworks: finite-state and functional dependency grammar*. PhD Thesis, University of Helsinki.

Voutilainen, A. (1999). Hand-crafted rules. In Van Halteren, H., editor, *Syntactic Wordclass Tagging*, pages 217–246. Springer.

Yli-Jyrä, A. (2005). *Contributions to the Theory of Finite-State Based Linguistic Grammars*. PhD Thesis, University of Helsinki.