ON THE COMPLEXITY OF SOME EXTENSIONS OF RCG PARSING

Eberhard Bertsch
Ruhr University
Faculty of Mathematics
Universitätsstraße 150
D-44780 Bochum
Germany
eberhard.bertsch@ruhr-uni-bochum.de

Mark-Jan Nederhof*
University of Groningen
Faculty of Arts
P.O. Box 716
NL-9700 AS Groningen
The Netherlands
markjan@let.rug.nl

Abstract

We consider the parsing problem for range concatenation grammars (RCGs). Two new applications of RCG parsing are studied. The first is the parsing of finite automata, the second is string-to-string transduction, with an extension of RCGs. We show that these problems are undecidable in general, but become tractable for subclasses of the formalism.

1 Introduction

Range concatenation grammar is a formalism that has a number of attractive theoretical and practical properties. First, the class of languages that can be described by RCGs (the range concatenation languages, or RCLs) equals the class PTIME of languages that can be recognized in polynomial time; RCGs share this property with a number of other formalisms such as ILFP [19].[1] This means that the generative power of RCGs includes that of e.g. context-free grammars, tree-adjoining grammars, and linear context-free rewriting systems.

Second, the combination of its expressiveness and its simplicity makes it attractive as intermediate representation in the construction of parsers for grammars in some other formalisms [4].

Third, it has been argued that it is suitable for description of a number of natural language phenomena [7].

Until now, studies of RCGs have concentrated on recognition of strings. However, most applications in NLP require more elaborate types of processing. The objective of this paper is to investigate two extensions of RCG recognition of strings: intersection with regular languages, and transduction both for strings and regular languages as input.

The structure of this paper is as follows. In the next section we recall the main definitions connected to RCGs, and we present a simple recognition algorithm for strings. In Section 3 we generalize this to the intersection with regular languages and show that for arbitrary RCGs, the emptiness problem for the intersection is undecidable. It is, furthermore, NP-complete if the regular language is finite and is represented by a finite automaton. We also show that the intersection can be computed in polynomial time provided we take an RCG satisfying a syntactic restriction.

*Supported by the Royal Netherlands Academy of Arts and Sciences. Secondary affiliation is the German Research Center for Artificial Intelligence (DFKI).

[1] Also worth mentioning are the formalisms from [23, 16], which describe languages that are related through logspace reductions to various other complexity classes besides PTIME.

The practical interest of this lies e.g. in the realm of spoken-language systems, where grammatical processing is often not applied on simple strings, but on a compact representation of many strings, in the form of a (weighted) finite automaton, which is output by a speech recognizer.

Another generalization of RCG parsing is range concatenation transduction, which is investigated in Section 4. We show that this type of transduction leads to undecidable problems for unconstrained RCGs. However, by imposing syntactic restrictions similar to the one in the section before, we obtain tractable algorithms.

Applications of transduction lie in machine translation, but transduction may also be used merely to specify more accurately the desired output format of a parser.

In Appendix A we show that the class of languages generated by RCGs indeed equals PTIME.

2 Range Concatenation Grammar

In this section we define positive range concatenation grammars. Since we do not consider negative RCGs in this paper, we will omit the qualification 'positive' before 'RCG'. For more details we refer to [9].

An RCG is a 5-tuple $G = (N, T, V, P, S)$ where N is a finite set of predicates, T and V are finite, disjoint sets of terminals and variables, respectively, $S \in N$ is the start predicate, and P is a finite set of clauses of the form:

$$\psi_0 \rightarrow \psi_1 \cdots \psi_m$$

where $m \geq 0$ and each of ψ_0, ψ_1, ..., ψ_m is a predicate expression of the form:

$$A(\alpha_1, \ldots, \alpha_p)$$

where $A \in N$, $p \geq 1$ is the arity of the predicate expression, and each argument α_i, $1 \leq i \leq p$, is an element of $(T \cup V)^*$. We assume each predicate occurs with only one arity. The start predicate has arity 1.

Lower-case letters $a, b, \ldots$ will denote terminals, upper-case letters $A, B, \ldots$ will denote predicates, and upper-case letters $X, Y, \ldots$ will denote variables. The empty string is denoted by ϵ. A symbol such as $\vec{\rho}$ will refer to a list of arguments in a predicate expression.

We say an RCG is a *simple* RCG (sRCG) if the arguments in the RHS of a clause consist of single variables, and each variable in a clause has exactly two occurrences: one in the RHS and one in the LHS.

An example of a (simple) RCG is given by the following clauses, which we have labelled for ease of reference:

$$\begin{aligned}
c_1: &\quad S(XY) \rightarrow A(X, Y) \\
c_2: &\quad A(Xa, Ya) \rightarrow A(X, Y) \\
c_3: &\quad A(Xb, Yb) \rightarrow A(X, Y) \\
c_4: &\quad A(\epsilon, \epsilon) \rightarrow \epsilon
\end{aligned}$$

As we will explain, this grammar generates the language $\{ww \mid w \in \{a, b\}^*\}$.

As for example in [3, 24], we consider the parsing problem as the combination of two steps. First, we construct a context-free grammar out of an RCG and an input string, in such a way that the CFG can be seen as a compact representation of all parses of the string. The second, optional step reduces the CFG, removing productions that cannot be part of any derivation.

The nonterminals of the context-free grammar have the form $A((r_1, r'_1), (r_2, r'_2), \ldots, (r_p, r'_p))$, for a predicate expression $A(\alpha_1, \ldots, \alpha_p)$ from some clause of the RCG, and a combination of numbers $r_1, r'_1, r_2, r'_2, \ldots, r_p, r'_p$, satisfying $0 \leq r_i \leq r'_i \leq n$ $(1 \leq i \leq p)$. In effect, we replace the arguments by pairs of numbers indicating *ranges*; ranges represent occurrences of substrings in the input. The start symbol is $S((0, n))$, where S is the start predicate from the RCG.

The productions of the CFG are formed as follows. For each clause:

$$\psi_0 \rightarrow \psi_1 \cdots \psi_m$$

we consider all possible mappings from occurrences of terminals and from variables in the clause to pairs of positions (ranges), satisfying:

- if an occurrence of terminal a is mapped to a pair (r, r'), then $a_{r+1} \cdots a_{r'}$ must be a, and
- if consecutive variables and occurrences of terminals in an argument are mapped to $(r_1, r'_1), (r_2, r'_2), \ldots, (r_q, r'_q)$, some q, then $r'_1 = r_2, r'_2 = r_3, \ldots, r'_{q-1} = r_q$. By definition, we then state that the above-mentioned mapping maps the argument as a whole to range (r_1, r'_q).

For each such mapping, we now add to the context-free grammar the production:

$$\psi'_0 \rightarrow \psi'_1 \cdots \psi'_m$$

where each ψ'_i $(0 \leq i \leq m)$ is obtained from ψ_i by replacing each argument by the range it is mapped to.

For the RCG from the running example and the input *abab*, we obtain productions such as $S((0,4)) \rightarrow A((0,2),(2,4))$ from clause c_1, $A((0,2),(2,4)) \rightarrow A((0,1),(2,3))$ from clause c_3, $A((0,1),(2,3)) \rightarrow A((0,0),(2,2))$ from clause c_2, and $A((0,0),(2,2)) \rightarrow \epsilon$ from clause c_4. We also obtain for example $A((3,3),(4,4)) \rightarrow \epsilon$ from clause c_4 but this production will never be used. No production $A((0,1),(3,4)) \rightarrow A((0,0),(3,3))$ will be produced, since the first and fourth symbols of the input *abab* are distinct, so neither c_2 nor c_3 can give rise to such a production.

If we now reduce the CFG, then we obtain a non-empty grammar if and only if some derivation of the empty string from $S((0, n))$ exists, and this defines whether the input string is in the language generated by the RCG.

In the running example, the reduction would remove e.g. $A((3,3),(4,4)) \rightarrow \epsilon$, and in the resulting grammar, the existence of the derivation $S((0,4)) \rightarrow A((0,2),(2,4)) \rightarrow A((0,1),(2,3)) \rightarrow A((0,0),(2,2)) \rightarrow \epsilon$ indicates that the input *abab* is indeed in the language generated by the grammar. At the same time, this derivation assigns a structure to the input. In this example, there is only one derivation. If there are several, the CFG serves to represent all of them in a compact way.

Since a context-free grammar can be reduced in linear time, the recognition and parsing problems for an RCG can be no more expensive than the construction of the (unreduced) context-free grammar from the RCG and an input string. This time complexity is (deterministic) polynomial in the length of the string, and linear in the size of the RCG. See [9] for more information on RCG parsing.[2]

It has been shown by [6] that the class of languages generated by simple RCGs equals the class of languages that can be described by the linear context-free rewriting systems, which is equal to the

[2] However, the algorithm as formulated in Table 1 of [9] treats cycles of clauses incorrectly. The author confirmed this in private communication, further claiming to have developed a correct version after publication. The published algorithm fails e.g. for the RCG with the clauses: $s(X) \rightarrow p(X)\ r(X)$, $p(X) \rightarrow q(X)$, $q(X) \rightarrow r(X)$, $q(a) \rightarrow \epsilon$, $r(X) \rightarrow p(X)$; the recognizer will erroneously reject input a.

classes of languages generated by the multiple context-free grammars [21] and the finite-copying LFGs [20].

3 Intersection with Regular Languages

It has been shown that the problem whether an RCG generates an empty language is undecidable [8]. Since $L = L \cap T^*$ for each RCL L, it is obvious that in general we cannot construct some description G' of a language $L' = L \cap R$, where L is generated by an RCG and R is a regular language, in such a way that the emptiness of L' can be decided on the basis of G'. In other words, the algorithm for parsing strings in polynomial time that we investigated in the previous section cannot be generalized to parsing *sets* of strings, if those sets are arbitrary regular languages.

We can simplify the task by constraining a regular language to be finite, which means that it is the language accepted by a cycle-free finite automaton, where we measure the time complexity in terms of the size of the automaton (number of transitions). Now the problem becomes decidable, but it is still NP-complete, which is proved by the following.

We can nondeterministically choose a string of length smaller than the size of the automaton. We can then check in polynomial time whether the string is accepted by the automaton and whether it is in the language generated by the RCG, using the algorithm from the previous section. This test succeeds for some string if and only if the intersection of the two languages is non-empty. Hence, our problem is in NP.

To finish our proof of NP-completeness, consider the NP-complete problem 3SAT. An instance of this problem consists of a collection $U = \{u_1, \ldots, u_n\}$ of variables and a collection $C = \{c_1, \ldots, c_m\}$ of clauses. If $u \in U$ then u and $\overline{u}$ are literals. A clause c_j $(1 \leq j \leq m)$ is a set $\{l_1, l_2, l_3\}$ of literals. A truth assignment is a function $t : U \rightarrow \{T, F\}$. For more explanation we refer to [11].

We will now define a polynomial algorithm to construct an RCG from an instance of the 3SAT problem, in such a way that the RCG accepts the set of truth assignments satisfying this instance. A truth assignment t will be encoded as a string $l_1 \cdots l_n$, where $l_i = u_i$ if $t(u_i) = T$ and $l_i = \overline{u}_i$ if $t(u_i) = F$, for $1 \leq i \leq n$.

The RCG will have $2n$ terminals, viz. u and $\overline{u}$, for each $u \in U$. For each variable $u_i \in U$ $(1 \leq i \leq n)$ we construct two RCG clauses $A_i(u_i) \rightarrow \epsilon$ and $A_i(\overline{u}_i) \rightarrow \epsilon$ and for each clause $c_j = \{l_1, l_2, l_3\}$ $(1 \leq j \leq m)$ we construct three RCG clauses $B_j(l_1) \rightarrow \epsilon$, $B_j(l_2) \rightarrow \epsilon$ and $B_j(l_3) \rightarrow \epsilon$ and the RCG furthermore contains the clause:

$$S(Y_1 \cdots Y_n) \;\rightarrow\; A_1(Y_1) \;\cdots\; A_n(Y_n)\; B_1(X_1) \;\cdots\; B_m(X_m)$$

Next, we construct a finite automaton that accepts the language $\{u_1, \overline{u}_1\} \times \ldots \times \{u_n, \overline{u}_n\}$ of all possible truth assignments. This can trivially also be done in polynomial time in the size of the instance of the problem. It follows that the intersection of the languages described by the RCG and the automaton is non-empty if and only if the instance of the problem has a solution. Thereby we have shown that deciding the emptiness problem for the intersection of RCLs and languages accepted by cycle-free finite automata is at least as difficult as deciding the 3SAT problem, and hence the former problem is NP-complete.

To allow emptiness of the intersection with regular languages to be decidable in polynomial time, we simplify the problem by constraining the RCG to be a simple RCG, and we apply an idea originally

due to [3]. Let us assume a regular language is given by a finite automaton with initial state q_0 and a set $\mathcal{F}$ of final states, and let us assume without loss of generality that there are no epsilon transitions.

For each clause in the sRCG we consider mappings from occurrences of terminals and from variables to ranges as before, but now the ranges are pairs of states of the automaton, and the first condition on such mappings is that if an occurrence of terminal a is mapped to a pair (r, r'), then there must be a transition labelled a from state r to state r'. The second condition remains as in Section 2.

As before, each such mapping gives rise to a production of a CFG. In addition, this CFG contains one production $S^\dagger \rightarrow S(q_0, q)$ for each final state $q \in \mathcal{F}$, and the new symbol $S^\dagger$ becomes the start symbol. The resulting CFG can then be reduced, as before. Thereby the emptiness problem can be decided in polynomial time in the size of the finite automaton.

4 Transduction

A *transduction* is a subset of $T_1^* \times T_2^*$, where T_1 is an input alphabet and T_2 is an output alphabet. A description of a transduction will be called a *transducer*. A transducer is a formal model of translation between two languages, which can be explicitly based on syntactic structures of varying complexity depending on the kind of machinery that is involved in the description.

Two finite automata that share some structural properties may be combined to form a finite transducer [5], and two similarly structured CFGs may be combined to form a syntax-directed translation schema [1].[3] By the same principle, we may combine two RCGs, and have them specify a transduction.

We call such a combination of two RCGs a *range concatenation transducer* (RCT). The difference from an RCG is that predicate expressions in clauses have the form:

$$A(\alpha_1, \ldots, \alpha_p)(\beta_1, \ldots, \beta_{p'})$$

where $A \in N$, $p \geq 1$ and $p' \geq 1$ are the input and output arities, and each input argument α_i $(1 \leq i \leq p)$ and each output argument β_i $(1 \leq i \leq p')$ is an element of $(T \cup V)^*$. We will assume that each variable occurs either in input arguments or in output arguments, but not in both.

An example of an RCT is given by the following clauses:

$$\begin{aligned}
&S(X_1Y_1)(X_2Y_2) \rightarrow A(X_1, Y_1)(X_2, Y_2)\\
&A(aX_1, Y_1a)(aX_2, aY_2) \rightarrow A(X_1, Y_1)(X_2, Y_2)\\
&A(bX_1, Y_1b)(bX_2, bY_2) \rightarrow A(X_1, Y_1)(X_2, Y_2)\\
&A(\epsilon, \epsilon)(\epsilon, \epsilon) \rightarrow \epsilon
\end{aligned}$$

The transduction it describes is $\{(ww^R, ww) \mid w \in \{a, b\}^*\}$. (The operator R in w^R reverses its argument string.) In other words, input palindromes are changed into output strings by reversing their second halves to produce two consecutive copies of the same string.

The meaning of an RCT is specified as follows. Given an input string $a_1 \cdots a_n$ we form a grammar by consistently replacing the input arguments by ranges, just as in the case of recognition as explained in Section 2, but now, because the output arguments remain unaffected by this process, the resulting grammar is an RCG. The predicate expressions in this RCG have the form

$$A((r_1, r_1'), (r_2, r_2'), \ldots, (r_p, r_p'))(\beta_1, \ldots, \beta_{p'})$$

[3] Consider also the combination of two tree-adjoining grammars to form a synchronous TAG [22]. The approach from [2] fits less well into this context, since it is based on tree transformations that are inherently asymmetric with respect to the relation between input and output.

where each (r_i, r'_i) indicates a range in the input string, as before, and $A((r_1, r'_1), (r_2, r'_2), \ldots, (r_p, r'_p))$ in its entirety represents a predicate in the resulting RCG. The set of strings that this RCG generates now represents the output of the transducer for input $a_1 \cdots a_n$.

The order of complexity of the above construction is identical to that of RCG recognition, viz. polynomial in the length of the input. However, the above construction merely produces an RCG that generates the set of output strings, but it does not include the process of actually finding any of those output strings. In fact, it is undecidable whether, for a given input string and a given RCT, the set of output strings is empty. We will prove an even stronger statement: there is a fixed RCT such that it is undecidable whether, for a given input string, the set of output strings is empty.

The idea of this proof is to let the input string represent an instance of Post's correspondence problem (PCP) [18]. For such an instance $\{(s_{1,1}, s_{1,2}), (s_{2,1}, s_{2,2}), \ldots, (s_{m,1}, s_{m,2})\}$, where $s_{i,1}, s_{i,2} \in \{a, b\}^*$ $(1 \leq i \leq m)$, the representation as input string is $\$s_{1,1}\#s_{1,2}\$s_{2,1}\#s_{2,2}\$...\$s_{m,1}\#s_{m,2}\$$.

The RCT is such that the set of output strings is non-empty if and only if the instance of PCP represented by the input has a solution. This RCT is given by the following set of clauses:

$$S(Z)(Y) \rightarrow \mathit{Post}(Z)(Y, Y)$$
$$\mathit{Post}(Z)(Y_1, Y_2) \rightarrow \mathit{Substrings}(Z)(Y_1, Y_2)$$
$$\mathit{Post}(Z)(X_1Y_1, X_2Y_2) \rightarrow \mathit{Post}(Z)(X_1, X_2) \quad \mathit{Substrings}(Z)(Y_1, Y_2)$$
$$\mathit{Substrings}(Z_1\$X_1\#X_2\$Z_2)(Y_1, Y_2) \rightarrow \mathit{Substring}(X_1)(Y_1) \quad \mathit{Substring}(X_2)(Y_2)$$
$$\mathit{Substring}(aX)(aY) \rightarrow \mathit{Substring}(X)(Y)$$
$$\mathit{Substring}(bX)(bY) \rightarrow \mathit{Substring}(X)(Y)$$
$$\mathit{Substring}(\epsilon)(\epsilon) \rightarrow \epsilon$$

Please note that the argument Z in $S(Z)(Y)$ represents the input, which encodes a specific instance of PCP, and Y represents a *match* for the problem instance (if one exists). Through the use of predicate *Substring* we ensure that pairs of strings contained in the instance of PCP are matched appropriately with subranges in the output Y.

Let us define the left and right projections of a transduction $\mathcal{T}$ to be the sets $\{w \mid (w, v) \in \mathcal{T}\}$ and $\{v \mid (w, v) \in \mathcal{T}\}$, respectively. In the case of the RCT for PCP above, the left projection is an undecidable language, and hence not in PTIME and not in the class of RCLs. This shows that the left projection of a range concatenation transduction need not be a range concatenation language, and by symmetry, the right projection need not be an RCL either.

We will now consider two simplified forms of RCT that allow tractable transduction. We first define a simple RCT (sRCT) as an RCT which is such that both the input and output arguments in the RHS of a clause consist of single variables, and each variable in a clause has exactly two occurrences: one in the RHS and one in the LHS. We define a right-simple RCT (rsRCT) as an RCT which is such that the above restriction holds only on the output arguments and the variables occurring therein.

Note that the left and right projections of a simple RCT are simple RCLs. The sRCGs generating these sRCLs are obtained by removing the output or input arguments, respectively, of the sRCT. Similarly, the left projection of a right-simple RCT is an RCL.

Using the transduction algorithm from the beginning of this section, we can produce an RCG from a right-simple RCT and an input string, in polynomial time, and this RCG is obviously also simple. The same holds for a simple RCT and a finite automaton as input, generalizing the construction from Section 3. Let us refer to the simple RCG that results in either of these cases as $\mathcal{G}$.

The second step is the reduction of the grammar $\mathcal{G}$, of which the time cost is linear in the size of $\mathcal{G}$, which is polynomial in the size of the input string or input automaton.

The next step is an analysis of $\mathcal{G}$, which will be used to avoid that the derivations we are going to compute contain cycles. We say an sRCG is *cyclic* if for some fixed string it allows derivations of unbounded length. This is a generalization of the definition of cyclicity for CFGs [1]. Our analysis is similar to that for the variant of Earley's algorithm from [12].

For the analysis of cycles, we first need to investigate which predicates of the sRCG are *nullable*. Nullable predicates are defined inductively as follows:

- If there is a clause $A(\vec{\rho}_0) \rightarrow B_1(\vec{\rho}_1) \cdots B_m(\vec{\rho}_m)$, with $m \geq 0$, such that $B_1, \ldots, B_m$ are all nullable, and the arguments in $\vec{\rho}_0$ do not contain any terminals, then A is also nullable.[4]

Similarly, *non-empty* predicates are defined inductively as follows:

- If there is a clause $A(\vec{\rho}_0) \rightarrow B_1(\vec{\rho}_1) \cdots B_m(\vec{\rho}_m)$, with $m \geq 0$, such that at least one of $B_1, \ldots, B_m$ is non-empty, or if $\vec{\rho}_0$ contains a terminal, then A is non-empty.

We now add new clauses by merging pairs of existing clauses. The motivation is that with these new clauses, we can avoid use of existing clauses in such a way that they might lead to cycles. The following is to be repeated until no more new clauses can be added.

- Consider two clauses $A_0(\vec{\rho}_0) \rightarrow A_1(\vec{\rho}_1) \cdots A_m(\vec{\rho}_m)$ and $B_0(\vec{\sigma}_0) \rightarrow B_1(\vec{\sigma}_1) \cdots B_{m'}(\vec{\sigma}_{m'})$ such that $A_i = B_0$, for some i $(1 \leq i \leq m)$, and $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m$ are all nullable, and $\vec{\rho}_0$ does not contain any terminals.
- Assume the sets of variables in these two clauses are disjoint. (If not, rename the variables in the first clause.)
- Construct $\vec{\tau}_0$ from $\vec{\rho}_0$, by replacing each variable that occurs in $\vec{\rho}_i$ by the corresponding argument in $\vec{\sigma}_0$, and by replacing each variable that occurs in $\vec{\rho}_1, \ldots, \vec{\rho}_{i-1}, \vec{\rho}_{i+1}, \ldots, \vec{\rho}_m$ by ϵ.
- Add the clause $A(\vec{\tau}_0) \rightarrow B_1(\vec{\sigma}_1) \cdots B_{m'}(\vec{\sigma}_{m'})$.

Note that each RHS of a new clause is the RHS of an existing clause from $\mathcal{G}$, and such a RHS contains exactly as many variables as the output arguments of the RHS of some clause from the original (r)sRCT. The predicate of the LHS of a new clause is an existing predicate from $\mathcal{G}$. The terminals in such a LHS occur together in the output arguments of the LHS of some clause from the original (r)sRCT, but they may be distributed in a different way over the respective arguments. Likewise, for a fixed RHS, each new clause for that RHS may have its LHS variables (which are the same as the variables in the RHS) distributed in a different way over the respective arguments. But since we consider the sRCT as fixed, the number of different distributions of variables and terminals over the arguments does not play any role in the complexity analysis of the growth of the grammar by the above transformation. Therefore, we only need to consider the RHSs and the LHS predicates. We conclude that the grammar after the above transformation has a size quadratic in the size it had before.

The final step is to show that we may effectively extract each output string from the resulting sRCG by a nondeterministic process, indicated in Figure 1. This process operates by recursive-descent. From

[4] Note that if $m = 0$ and $\vec{\rho}_0$ does not contain any terminals, then $A(\vec{\rho}_0)$ is of the form $A(\epsilon, \ldots, \epsilon)$, since the grammar is a *simple* RCG.

Procedure **generate**:

1. Nondeterministically choose from steps 2 and 3 as far as they are applicable.
2. Applicable if the start predicate is non-empty: Let the return value be the string in the result of calling routine **generate-from** with argument S.
3. Applicable if the start predicate is nullible: Let the return value be ϵ.

Procedure **generate-from** with argument A_0:

1. Choose:
   - a clause $A_0(\vec{\rho}_0) \rightarrow A_1(\vec{\rho}_1) \cdots A_m(\vec{\rho}_m)$, and
   - two disjoint, complementing subsequences $B_1(\vec{\rho'}_1) \cdots B_{m'}(\vec{\rho'}_{m'})$ and $C_1(\vec{\rho''}_1) \cdots C_{m''}(\vec{\rho''}_{m''})$ of $A_1(\vec{\rho}_1) \cdots A_m(\vec{\rho}_m)$ (i.e. $m' + m'' = m$), where $B_1, \ldots, B_{m'}$ are all non-empty and $C_1, \ldots, C_{m''}$ are all nullible,

   such that $\vec{\rho}_0$ contains one or more terminals or $m' \geq 2$.
2. Call the procedure **generate-from** recursively with argument $B_1, \ldots, B_{m'}$, respectively, and let the returned tuples of strings be $\vec{\sigma}_1, \ldots, \vec{\sigma}_{m'}$.
3. Construct $\vec{\sigma}_0$ from $\vec{\rho}_0$ by consistently replacing each variable that occurs in some $\vec{\rho'}_i$ by the corresponding string in $\vec{\sigma}_i$ $(1 \leq i \leq m')$, and by replacing each variable that occurs in $\vec{\rho''}_1, \ldots, \vec{\rho''}_{m''}$ by ϵ.
4. Let the return value be $\vec{\sigma}_0$.

Figure 1: Computing an output string from sRCG $\mathcal{G}$.

the start predicate, we descend the grammar, finding tuples of strings for RHS predicate expressions that we concatenate into tuples of larger strings for LHS predicate expressions.

It can be easily seen that the number of incarnations of procedure **generate-from** is linear in the length of the output string, since each incarnation is responsible either for the generation of a terminal, if the LHS of the clause contains a terminal, or it is responsible for the joining of tuples obtained recursively for members in the RHS of the clause, such that at least two of these tuples each contain at least one non-empty substring. The number of incarnations can thereby not exceed twice the length of the output string.

Apart from the use of $\mathcal{G}$ for nondeterministic generation of output strings, it may also be used to decide the emptiness and membership problems for the output language, as follows. The output language is empty if and only if $\mathcal{G}$ becomes the empty grammar after reduction. Membership of a particular string in the output language can be decided by applying the general recognition algorithm for RCGs from Section 2.

Complexity theory provides us with many language classes that are characterized through automaton models and their restrictions by time and space bounds. One such important class is PTIME, and we know that this class is exactly the class of languages generated by RCGs.

Less seems to be known about characterizations of classes of *transductions* through complexity measures. Although simple RCTs and right-simple RCTs have apparently favourable complexity properties, it seems difficult to capture these properties in formal terms. The algorithm in Figure 1 can for example be described in terms of some appropriate nondeterministic RAM model with linear time bounds [13], but since such models allow NP-complete languages, it seems they are too course

for our purposes.

From an algorithmic point of view, the most obvious implementation of a transduction is as a test of membership for pairs, each consisting of an input and an output string. In this section we have chosen another approach whereby an input string is considered in isolation and processed to produce the grammar $\mathcal{G}$, to be used in a later phase for identifying matching output strings. This amounts to a form of preprocessing for the above-mentioned membership test, and thereby the algorithms in this section can be seen in the light of some theory of compilability [10]. Further research is needed to decide whether application of such theory may lead to more accurate characterizations of types of range concatenation transduction in terms of automaton models and complexity measures.

5 Conclusions

The class of range concatenation grammars introduced by Pierre Boullier was previously shown to provide ease of expression in descriptions of linguistic phenomena and to allow, by means of natural subclasses, comparison with other formalisms known from the computational linguistics literature. Furthermore, the complexity class PTIME known from theoretical computer science is identical with the class of RCLs, as demonstrated in the appendix to this paper. The main contribution of the present article consists in extending the RCG concept in two directions that seem natural from the point of view of phrase-structure grammars, but have not been studied in an RC context before: parsing of representations of regular languages, and grammar-based transduction. The essential results can be summarized by stating that both problems become tractable by constraining grammars to be 'simple'.

Acknowledgements

Giorgio Satta contributed to the proof in Appendix A, and made several helpful remarks on the core ideas of the paper. Pierre Boullier kindly provided references to publications.

References

[1] A.V. Aho and J.D. Ullman. *Parsing*, The Theory of Parsing, Translation and Compiling, volume 1. Prentice-Hall, 1972.

[2] B.S. Baker. Generalized syntax directed translation, tree transducers, and linear space. *SIAM Journal on Computing*, 7(3):376–391, 1978.

[3] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. In Y. Bar-Hillel, editor, *Language and Information: Selected Essays on their Theory and Application*, chapter 9, pages 116–150. Addison-Wesley, 1964.

[4] F. Barthélemy et al. Guided parsing of range concatenation languages. In *39th Annual Meeting and 10th Conference of the European Chapter of the ACL*, pages 42–49, 2001.

[5] J. Berstel. *Transductions and Context-Free Languages.* B.G. Teubner, Stuttgart, 1979.

[6] P. Boullier. Proposal for a natural language processing syntactic backbone. Rapport de recherche 3342, INRIA, Rocquencourt, France, January 1998.

[7] P. Boullier. Chinese numbers, MIX, scrambling, and Range Concatenation Grammars. In *Ninth Conference of the European Chapter of the ACL*, pages 53–60, 1999.

[8] P. Boullier. A cubic time extension of context-free grammars. In *Sixth Meeting on Mathematics of Language*, pages 37–50, Orlando, Florida USA, July 1999. University of Central Florida. Also appeared in *Grammars*, 3:111-131, 2000.

[9] P. Boullier. Range Concatenation Grammars. In *Proceedings of the Sixth International Workshop on Parsing Technologies*, pages 53–64, Trento, Italy, February 2000.

[10] M. Cadoli et al. Preprocessing of intractable problems. DIS 24-97, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", November 1997. To appear in Information and Computation, 2001(?).

[11] M.R. Garey and D.S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[12] S.L. Graham, M.A. Harrison, and W.L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[13] E. Grandjean. Linear time algorithms and NP-complete problems. *SIAM Journal on Computing*, 23(3):573–597, 1994.

[14] A. Groenink. *Surface without Structure – Word order and tractability issues in natural language analysis*. PhD thesis, University of Utrecht, 1997.

[15] A.V. Groenink. Mild context-sensitivity and tuple-based generalizations of context-grammar (sic). *Linguistics and Philosophy*, 20:607–636, 1997.

[16] N. Immerman. Relational queries computable in polynomial time. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 147–152, 1982.

[17] K.N. King. Alternating multihead finite automata. *Theoretical Computer Science*, 61:149–174, 1988.

[18] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[19] W.C. Rounds. LFP: A logic for linguistic descriptions and an analysis of its complexity. *Computational Linguistics*, 14(4):1–9, 1988.

[20] H. Seki et al. Parallel multiple context-free grammars, finite-state translation systems, and polynomial-time recognizable subclasses of lexical-functional grammars. In *31st Annual Meeting of the ACL*, pages 130–139, 1993.

[21] H. Seki et al. On multiple context-free grammars. *Theoretical Computer Science*, 88:191–229, 1991.

[22] S.M. Shieber. Restricting the weak-generative capacity of synchronous tree-adjoining grammars. *Computational Intelligence*, 10(4):371–385, 1994.

[23] M.Y. Vardi. The complexity of relational query languages. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[24] K. Vijay-Shanker and D.J. Weir. The use of shared forests in tree adjoining grammar parsing. In *Sixth Conference of the European Chapter of the ACL*, pages 384–393, 1993.

A The Class of RCLs Equals PTIME

In Section 2 we saw that the RCLs are included in PTIME. For the converse result, that each language in PTIME is generated by an RCG, no complete proof has yet appeared in print. RCG and ILFP differ substantially so that the proof from [19] cannot be easily adapted to RCG, and a proof in [14, Chapter 5] for simple LMG, which is closely related to RCG, is incomplete and also the proof in [15] is far from explicit; furthermore, the proof of equivalence of simple LMG and iLFP via iLMG in [14, pp. 102 and 106] does not suffice either since Groenink's iLFP differs from Rounds' ILFP in a number of respects. Instead of revising any of the above proofs, we give a new proof that does not consider any other grammatical formalism than RCG.

Our proof consists of two steps. First, in [17] it is shown that PTIME equals the class of languages accepted by two-way alternating finite automata with k heads.[5] Second, we show that for any two-way alternating finite automaton with k heads, we can construct an equivalent RCG. In this RCG, existential states are modelled by having several clauses for a predicate and universal states are modelled by having a conjunction of predicate expressions in the RHS of a single clause. The RCG has 2^k predicates per state, and in addition there is a start predicate S, a predicate *symbol* defined by one clause $symbol(a) \rightarrow \epsilon$ for each symbol $a \in T$, and a predicate *equal* defined by $equal(X, X) \rightarrow \epsilon$.

A predicate for a state q will have arity $k+1$ and will have the form $q^{\vec{b}}$, where $\vec{b} \in \{0,1\}^k$. The use of such a predicate in a predicate expression represents a configuration of the automaton. Each of the k symbols of $\vec{b}$ is 1 if and only if the corresponding tape head would be positioned on the start-of-sentence marker ¢. Each of the first k arguments will represent a suffix of the input, i.e. a range (i, n), indicating the following properties of the corresponding tape head:

- If the corresponding symbol from $\vec{b}$ is 0, then the tape head would be on position $i+2$. This means the tape head would be positioned on the end-of-sentence marker \$ if $i = n$.
- If the corresponding symbol from $\vec{b}$ is 1, then the tape head would be positioned on the first symbol of the tape, i.e. the start-of-sentence marker ¢. (This will only occur if $i = 0$.)

The last argument will be the complete input string (i.e. the range $(0, n)$) throughout.

There is one clause defining the start predicate:

$$S(X) \rightarrow q_0^{1^k}(\overbrace{X, \ldots, X}^{k}, X)$$

where q_0 is the initial state of the automaton. This indicates that all tape heads are initially on the start-of-sentence marker ¢.

[5] In an alternating automaton, a given state can be either "existential" or "universal". The former notion expresses the necessity of continued computation via *one* of the available successor configurations, which amounts to traditional nondeterminism, while the latter requires successful computation via *all* such configurations. The type of alternating finite automaton considered here has access to a read-only tape through k heads that can each move independently in both directions. The tape contains the start-of-sentence marker ¢, followed by the input, followed by the end-of-sentence marker \$.

We recall that a two-way alternating finite automaton with k heads has a head selector function τ that maps the current state p to a tape number j. The j-th tape head reads a symbol a and the transition function δ is applied on the pair (p, a) to yield a set of pairs (q, d), where q is a next state and $d \in \{-1, 0, 1\}$ encodes a movement of the j-th tape head.

For each existential state p, each $a \in T$, and each $\vec{b} = b_1 \cdots b_k \in \{0,1\}^k$ such that $\tau(p) = j$ and $b_j = 0$, and for each pair $(q, d) \in \delta(p, a)$ such that $d = 0$, the grammar has one clause of the form:

$$p^{\vec{b}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X) \rightarrow q^{\vec{b}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X)$$

For each pair $(q, d) \in \delta(p, a)$ such that $d = 1$, we have an identical clause except that the RHS argument expression contains X_j instead of aX_j, to indicate the tape head has shifted one position to the right. If $d = -1$, then the grammar contains the clauses:

$$p^{\vec{b}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X) \rightarrow \mathit{symbol}(Y) \;\; q^{\vec{b}}(X_1, \ldots, X_{j-1}, YaX_j, X_{j+1}, \ldots, X_k, X)$$

$$p^{\vec{b}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X) \rightarrow \mathit{equal}(aX_j, X) \;\; q^{\vec{c}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X)$$

where $\vec{c}$ is constructed from $\vec{b}$ by changing the j-th symbol to 1, which indicates the j-th tape head is now positioned on the start-of-sentence marker. Note that if aX_j is equal to X, which represents the entire input string, then this means that the tape head would be on the second position just before applying the transition.

For each existential state p, $a = \$$, and each $\vec{b} = b_1 \cdots b_k \in \{0,1\}^k$ such that $\tau(p) = j$ and $b_j = 0$, and for each pair $(q, d) \in \delta(p, a)$ such that $d = 0$, the grammar has one clause of the form:

$$p^{\vec{b}}(X_1, \ldots, X_{j-1}, X_j, X_{j+1}, \ldots, X_k, X) \rightarrow \mathit{equal}(X_j, \epsilon) \;\; q^{\vec{b}}(X_1, \ldots, X_{j-1}, X_j, X_{j+1}, \ldots, X_k, X)$$

The case $a = \$$, $b_j = 0$ and $d = -1$ is left to the imagination of the reader. For the case $a = ¢$ and $b_j = 1$, we have the clause:

$$p^{\vec{b}}(X_1, \ldots, X_{j-1}, X_j, X_{j+1}, \ldots, X_k, X) \rightarrow q^{\vec{c}}(X_1, \ldots, X_{j-1}, X_j, X_{j+1}, \ldots, X_k, X)$$

where $\vec{c} = \vec{b}$ if $d = 0$, and if $d = 1$ then $\vec{c}$ is constructed from $\vec{b}$ by changing the j-th symbol to 0.

For each universal state p, each $a \in T$, and each $\vec{b} = b_1 \cdots b_k \in \{0,1\}^k$ such that $\tau(p) = j$ and $b_j = 0$, the grammar has two clauses, one clause to model the case that the j-th tape head is at least two positions away from the start-of-sentence marker ¢, and one for the case that the head is immediately next to ¢. Both clauses have the same LHS, which is as in the case of existential states. The RHS of the first clause starts with $\mathit{symbol}(Y)$ and the second with $\mathit{equal}(aX_j, X)$. For each $(q, d) \in \delta(p, a)$, each of the two RHSs has one further predicate expression. For $d = -1$, this is $q^{\vec{b}}(X_1, \ldots, X_{j-1}, YaX_j, X_{j+1}, \ldots, X_k, X)$ for the first, and $q^{\vec{c}}(X_1, \ldots, X_{j-1}, aX_j, X_{j+1}, \ldots, X_k, X)$ for the second clause, where $\vec{c}$ is as for existential states. Other cases are left to the imagination of the reader.

If a state p is final, then for all $\vec{b} \in \{0,1\}^k$, the grammar also contains the clause:

$$p^{\vec{b}}(X_1, \ldots, X_k, X) \rightarrow \epsilon$$

This concludes the proof.