

PROGRAMMED GRAMMARS WITH RULE QUEUES

HENNING FERNAU*

Fachbereich IV, Abteilung Informatik, Universität Trier, D-54286 Trier, Germany

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We generalise the well-known notion of programmed grammars without appearance checking by introducing a buffer that could store sequences of rules, thus not necessarily describing the rule to be selected in the very next step, but rather in some future time.

1. Introduction and Motivation

Programmed grammars are a quite established device for describing formal languages. However, there are some important questions that are still open today even for the simplest model, programmed grammar without appearance checking, regarding closure properties: Is the corresponding language class closed under Kleene plus or star, or under (non-erasing) substitution? In short, it is still unknown whether these devices lead to abstract families of languages (AFL).

A possible venue to solve these problems could be to try to define natural extensions of these mechanisms that yield AFL. In this note, we propose and discuss one possible extension.

We expect the reader to be familiar with the usual notions in formal language theory. We will follow the convention that two languages are equivalent iff they differ by at most the empty word λ .

2. Definitions and First Results

A *programmed grammar without appearance checking* [1, 4] is a construct $G = (V_N, V_T, P, S)$, where V_N , V_T , and S are the set of nonterminals, the set of terminals and the start symbol, respectively, and P is a finite set of productions of the form $(r : \alpha \rightarrow \beta, \sigma(r))$, where $r : \alpha \rightarrow \beta$ is a rewriting rule labelled by r and $\sigma(r)$ (*success field*) is a set of labels of such core rules in P . By $\text{Lab}(P)$ we denote the set of all labels of the productions appearing in P . Mostly, we identify $\text{Lab}(P)$ with P . For (x, r_1) and (y, r_2) in $V_G^* \times \text{Lab}(P)$, (as usual, $V_G = V_T \cup V_N$ denotes the

*fernau@uni-trier.de

total alphabet) we write $(x, r_1) \Rightarrow (y, r_2)$ iff

$$x = z_1\alpha z_2, y = z_1\beta z_2, (r_1 : \alpha \rightarrow \beta, \sigma(r_1)) \in P, \text{ and } r_2 \in \sigma(r_1)$$

The language generated by G is defined as

$$L(G) = \{w \in V_T^* : (S, r_1) \xRightarrow{*} (w, r_2) \text{ for some } r_1, r_2 \in \text{Lab}(P)\}.$$

$\mathcal{L}(P, \text{CF})$ denotes the language class that can be described with programmed grammars without appearance checking with context-free core rules. We write $\mathcal{L}(P, \text{CF}-\lambda)$ to (possibly) restrict this class further by disallowing erasing rules.

Due to the connection to the Petri net reachability problem as exhibited in [2], the membership problem is decidable, i.e., the following is known:

Theorem 1 $\mathcal{L}(P, \text{CF}) \subseteq \mathcal{L}(\text{REC}) \subset \mathcal{L}(\text{RE})$.

We generalise this classical notion as follows:

A *programmed grammar (without appearance checking) with rule queues* is also a construct $G = (V_N, V_T, P, S)$, where V_N, V_T , and S are the set of nonterminals, the set of terminals and the start symbol, respectively, and P is a finite set of productions of the form $(r : \alpha \rightarrow \beta, \sigma(r))$, where $r : \alpha \rightarrow \beta$ is a rewriting rule labelled by r , but $\sigma(r)$ (*success field*) is now a non-empty, finite set of sequences of labels of such core rules in P . i.e., $\sigma(r) \subseteq (\text{Lab}(P))^*$, $|\sigma(r)| < \infty$.

For (x, s) and (y, t) in $V_G^* \times (\text{Lab}(P))^*$, we write $(x, s) \Rightarrow (y, t)$ iff, for some $r_1, r_2 \in \text{Lab}(P)$,

$$x = z_1\alpha z_2, y = z_1\beta z_2, (r_1 : \alpha \rightarrow \beta, \sigma(r_1)) \in P, \text{ and } r_2 \in \sigma(r_1), s = r_1 u, t = u r_2$$

Observe that the string of control labels is used in a queue fashion; in particular, this means:

- if $\sigma(r)$ only contains strings of length one, then programmed grammars with queues work in the same way as usual programmed grammars, but
- control information could be carried over relatively long distances between rules in a derivation sequence by first using $r_2 \in \sigma(r_1)$ with $|r_2| \geq 2$ and later using $\lambda \in \sigma(r_1)$ to shrink the queue again.

There are two natural ways to obtain languages: with empty queues or with arbitrary queues at the end of a derivation of a terminal string. Formally, this means:

$$L_{EQ}(G) = \{w \in V_T^* : (S, r_1) \xRightarrow{*} (w, \lambda) \text{ for some } r_1 \in \text{Lab}(P)\}.$$

$$L(G) = \{w \in V_T^* : (S, r_1) \xRightarrow{*} (w, r_2) \text{ for some } r_1 \in \text{Lab}(P), r_2 \in (\text{Lab}(P))^*\}.$$

So, analogously to programmed grammars, we obtain now four (interesting) language classes, denoted by $\mathcal{L}(\text{PEQ}, \text{CF})$ and $\mathcal{L}(\text{PEQ}, \text{CF}-\lambda)$ when terminating with the empty queue and $\mathcal{L}(\text{PQ}, \text{CF})$ and $\mathcal{L}(\text{PQ}, \text{CF}-\lambda)$ when terminating with arbitrary queues.

Let us first demonstrate that the proposed extension is interesting after all. Consider the grammar G with the following rules (and terminal alphabet $\{a\}$):

$(r_1 : S \rightarrow AA, \{r_2r_2\})$, $(r_2 : A \rightarrow S, \{r_1, r_3\})$, $(r_3 : S \rightarrow a, \{r_4\})$, $(r_4 : F \rightarrow F, \{\lambda\})$.

So, $(S, r_3) \Rightarrow (a, r_4)$, and $(S, r_1) \Rightarrow (AA, r_2r_2) \Rightarrow^2 (SS, s)$ with $s \in \{r_1, r_3\}^2$.

If $s = r_1r_1$, then we can restart the cycle, i.e. $(SS, r_1r_1) \Rightarrow^2 (AAAA, r_2r_2r_2r_2) \Rightarrow (SSSS, t)$ with $t \in \{r_1, r_3\}^4$.

If $s = r_3r_3$, then $(SS, r_3r_3) \Rightarrow^2 (aa, r_4r_4)$.

If $s = r_1r_3$, then $(SS, r_1r_3) \Rightarrow (AAS, r_3r_2r_2) \Rightarrow (AAa, r_2r_2r_4) \Rightarrow^2 (SSa, r_4r_1r_1)$.

Now, the derivation dies, since the rule with label r_4 is not applicable.

A similar problem arises for $s = r_3r_1$.

Inductively, one can show that (S^k, s) is a possible intermediate configuration of the grammar only if $s \in \{r_1, r_3\}^k$, but then, only $s = r_1^k$ or $s = r_3^k$ may lead to terminal strings in the end.

Moreover, another inductive argument can be used to show that

$$L = \{a^{2^n} \mid n \geq 0\} \subseteq L(G).$$

This reasoning altogether shows that $L = L(G)$. Since $L \notin \mathcal{L}(P, CF)$ is well-known (see [2]), we can conclude:

Lemma 1 $\mathcal{L}(P, CF-\lambda) \subsetneq \mathcal{L}(PQ, CF-\lambda)$ and $\mathcal{L}(P, CF) \subsetneq \mathcal{L}(PQ, CF)$.

3. Generative Power

We have seen that the proposed extension actually broadens the class of languages that can be described. We will see in this section that this is not an accident.

It is well-known that finite automata equipped with one queue as additional storage medium can recognise every recursively enumerable language, see [3, 5, 6, 7]. Formally, commands of such a queue automaton are of the form (q, a, q', w) , which means that the automaton, being in state q , reads symbol a from the left end of the queue (which would be erased after reading), switches to state q' and appends w to the right end of the queue. A queue automaton A can be specified as $A = (V_T, S, Q, \delta, q_0, F)$, where V_T is the input alphabet, $S = \{s_1, \dots, s_r\} \supset V_T$ is the queue alphabet, Q is the state alphabet, $\delta \subset Q \times S \times Q \times S^*$ is the finite transition relation, q_0 is the start state and F is the set of accepting states.

Theorem 2 $\mathcal{L}(PQ, CF-\lambda) = \mathcal{L}(PEQ, CF-\lambda) = \mathcal{L}(PQ, CF) = \mathcal{L}(PEQ, CF) = \mathcal{L}(RE)$.

Proof. (Sketch) As it is easy to see, we can assume that the given queue automaton $A = (V_T, S, Q, \delta, q_0, F)$ (with $S \cap Q = \emptyset$) will accept with a copy of the input word $w \in V_T^*$ as its only queue content, meaning that there is a disjoint copy V_T' of the input alphabet V_T and a morphism h given by $h(a) = a'$ such that $\#h(w)\dagger$ will be the queue content upon termination (except from the acceptance of the empty word), where $\#$ is a special symbol that will only appear in any queue in this context. For example, we can modify any given queue automaton by letting it copy the input word in the very beginning, yielding a queue content $w\#h(w)\dagger$, so that the simulating queue automaton simply ignores (skips) the suffix $\#h(w)\dagger$ in the following computation.

We produce a P(E)Q-grammar G with rule labels from $S \cup Q$ for the given RE-language $L(A) \subseteq V_T^+$ as follows^a:

(1) In a first phase, G generates an arbitrary word $w \in V_T^*$ in the queue; more precisely, G has the following rule:

(\uparrow : $S \rightarrow S, V_T^2 \cup \{\$\} \cup \{\uparrow\uparrow\} \cup \{\$\}V_T$).

\uparrow is a dummy symbol that is only used to build up the input within the queue. $\$$ is a delimiter to start the second phase that actually simulates the queue automaton, it should be introduced only once after having built up about half of the length of the input. Schematically, this phase works as follows: First, an arbitrary string \uparrow^m is put into the queue. Then, \uparrow^m yields (in one step) $\uparrow^{m-1} \$$ or $\uparrow^{m-1} \$a$ for some $a \in V_T$ in the queue. Thus, finally either a word from $\{\$\}V_T^{2m-2}$ or from $\{\$\}V_T^{2m-1}$ is found in the queue.

(2) ($\$$: $S \rightarrow q_0, \{\lambda\}$); (a : $q \rightarrow q', \{w \mid (q, a, q', w) \in \delta\}$) for all $(q, a, q', w) \in \delta$.

Finally, in a termination phase, we will actually generate the word accepted by the queue automaton:

(3) ($\#$: $q \rightarrow T, \{\lambda\}$) for $q \in F$,

(a' : $T \rightarrow aT, \{\lambda\}$) for all $a \in V_T$,

(\dagger : $T \rightarrow \lambda, \{\lambda\}$).

Notice that the grammar as described will generate the same language, irrespectively of whether it is interpreted in the EQ or in the Q mode due to the special right delimiter \dagger .

Finally, observe that a core rule of the form $A \rightarrow \lambda$ is used only once in the construction, so that by the known closure properties of the recursively enumerable languages, we could as well restrict ourselves to non-erasing grammars. \square

4. Discussion

We discussed a seemingly small modification of programmed grammars without appearance checking in order to define a language family that contains $\mathcal{L}(P,CF)$ but forms an AFL. In fact, we ended up with an AFL, but this AFL turned out to be well-known: the class of recursively enumerable languages. Possibly surprisingly, this is also true if we consider non-erasing core rules only. In fact, we are only using right-linear rules in our simulation, so that we can even state:

Corollary 1 $\mathcal{L}(PQ,REG)$ characterizes $\mathcal{L}(RE)$.

Could we possibly define further restrictions to obtain smaller language classes? One natural restriction would be to disallow arbitrarily large (auxiliary) computations on the queue. It is not hard to see that one can characterise the context-sensitive languages by requiring that the queue should never be longer than the terminal word that is going to be derived by the grammar. This criterion could be further simplified (for PEQ grammars without erasing rules) by allowing the empty word in the success field only within such rules that derive a terminal word. For such restricted devices, derivations that produce queues that are longer than the envisaged terminal word could never be successful. This would prevent the (mis)use

^aRecall that, by convention, we can restrict ourselves to λ -free RE-languages.

of the queue for all the computation. Our previous example displays the power of this more restricted version. Queue automata that work in the sketched way are discussed in [7]. In this context, it is also worth mentioning that the language class accepted by quasi-realtime queue automata forms an AFL, see [6].

Another idea would be to use the regulation string as a pushdown store, not as a queue.^b It is not hard to see that L^* for any $L \in \mathcal{L}(\text{P,CF})$ could be described in this way. It might be interesting to further study the corresponding language classes.

Acknowledgement We gratefully acknowledge discussions with Klaus Reinhardt on queue automata.

References

1. J. Dassow and Gh. Păun. *Regulated Rewriting in Formal Language Theory*, volume 18 of *EATCS Monographs in Theoretical Computer Science*. Springer, 1989.
2. D. Hauschildt and M. Jantzen. Petri net algorithms in the theory of matrix grammars. *Acta Informatica*, 31:719–728, 1994.
3. M. Li, L. Longpré, and P. M. B. Vitányi. The power of the queue. *SIAM Journal on Computing*, 21(4):697–712, 1992.
4. D. J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the ACM*, 16(1):107–131, 1969.
5. J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
6. B. Vauquelin and F. P. Zannettacci. Automates á file. *Theoretical Computer Science*, 11:221–225, 1980.
7. R. Vollmar. Über einen Automaten mit Pufferspeicherung. *Computing*, 5:57–70, 1970.

^bNotice that the results from [5] should not be misinterpreted: The storage structure that Shepherdson and Sturgis call “push down” is nowadays called a queue.