Proceedings of the Federated Conference on Computer Science and Information Systems pp. 711–720
DOI: 10.15439/2017F249
ISSN 2300-5963 ACSIS, Vol. 11

Properties and Limits of Supercombinator Set Acquired from Context-free Grammar Samples

Michal Sičák, Ján Kollár
Technical University of Košice, Department of Computers and Informatics
Letná 9, 042 01 Košice, Slovakia
Email: {michal.sicak, jan.kollar}@tuke.sk

***Abstract*—We present an improved version of algorithm that can transform any context-free grammar into a supercombinator form. Such a form is composed only of lambda calculus' supercombinators that are enriched by grammar operations. The main properties of this form are non-redundancy and scalability. We show the improvements that we've made to create smaller supercombinator set than in our previous algorithm's version. We present experiments performed on Context-free grammars obtained by transformation from Groningen meaning bank corpus. Experiments confirm that our form has a theoretical maximum limit of possible supercombinators. That limit is a mathematical sequence called Catalan number. We show that in some cases we are able to reach that limit if we use large enough input data source and we limit the size of supercombinator permitted into the final set. We also describe another benefit of our algorithm, which is the identification of most reoccurring structures in the input set.**

I. Introduction

LAMBDA calculus is a formalism that describes computation with the use of expressions, variables and applications. Combinators are lambda expressions without free variables. We use more restricted form of combinators, supercombinators in our work. The term supercombinator was coined by Hughes in [1] and it means an expression that can contain only constants or another supercombinators. In this paper we show an algorithm that can transform input grammar or a set of grammars into a single supercombinator form that is non-redundant yet retains the descriptive ability of its input grammars.

The main fuel of our work are grammars. We can use them for purposes, which exceed their usual application like the description of a language. The possibilities of wide grammar usage has been presented by Klint et al. in [2]. They argue that grammars are a strong formalism method that are already used in many areas of software engineering. We have presented in our work [3] a way to use grammars as a prime object of internal language incremental evolution.

We have shown in our previous work [4] that any Context-free grammar (CFG) can be transformed into a supercombinator form. Which means that we can abstract the structure from the data (represented in grammars as terminal symbols). The experiment performed in mentioned work showed that we can reduce the amount of grammar elements with this approach rather significantly. We have parsed samples of natural language with the Sequitur [5] algorithm and then converted resulting grammars into a supercombinator form. We have proven that our algorithm abstracts CFGs rather well. In this paper we are using a source that comes from a more meaningful background, short newspaper articles that have already been parsed with the use of Combinatory Categorial Grammars (CCG). We need a large enough data source that can be converted to CFG form for further processing. However, we do not process those data for some semantic related purpose. Our goal is not to create new meaning parser, but to analyze the possibilities of a CFG abstraction, to explore their structure and even contribute to the field of grammar metrics. Our ultimate goal is to create single scalable supercombinator structure that contains data non-redundantly.

The main contributions of this paper are:

- We present updated algorithm for supercombinator form acquisition that runs more smoothly than the one from our previous work [4]. Short description of its basic functionality and a list of performed changes are shown in the section III. We also explain there, why are those changes beneficial for the entire process. The improvements of our algorithm are described in the section III-D.
- We describe various experiments that we have performed on 62 008 grammar samples taken from 10 000 short newspaper articles included in the Groningen Meaning Bank (GMB) [6] corpus. We show in the section IV that growth of our supercombinator form is limited by a mathematical sequence called Catalan number. The achieved grammar element reduction performed on GMB inputs is still significant, as it was in our previous work where we have used Sequitur generated grammars.
- We show that supercombinators that have been merged to achieve non-redundancy can be tracked during that merge operation in order to acquire more information on the input data themselves. The results in the section IV-D show that we can identify the most reoccurring structures in the input form, as the structure is directly translated into supercombinators.

II. Motivation

One of motivations behind our work is the ability to process data stream of grammars at the input. As a grammar needs not to be predefined, it can be acquired from a plain text

This work was supported by project KEGA 047TUKE-4/2016 "Integrating software processes into the teaching of programming".

by a process called grammar inference. It is a well studied problem. We know that we cannot infer a grammar from a set of positive samples purely algorithmically. That has been proven by Gold in [7]. Usually in the communication we do not possess the knowledge about what is and what is not a correct sentence. There exist researches that study this phenomena in human to human communication. For example Onnis, Waterfall and Edelman found out in [8] that people, and especially little children, use cues called variation sets to differentiate utterances as grammatical or not. In the realm of formal languages, by using heuristics like statistical analysis or evolutionary algorithms, we can infer a grammar from positive samples with certain proficiency, see Stevenson and Cordy [9]. And such grammars might have a form of a CFG, therefore we can apply our process on them and obtain highly parallel, non-redundant structure that is scalable. We describe experiments in the section IV that give out the evidence to these claims.

One of the other reasons why we have created this process is to battle the phenomena called structural explosion [10]. This occurs for example when we are trying to create a finite state automaton from a regular expression that contains structurally identical parts. Let's have the expression (1) as an example.

$$a(b)^* \,|\, c(d)^* \,|\, e(f)^* \tag{1}$$

Three parts of that expression located in between alternative operators | are structurally identical, yet carry different symbols. We can see in Fig. 1 how the structural identity is reflected in the automaton created from the expression (1). Each strand representing identical structures has its own state and transitions. Although we can see that they are structurally identical, they are still fully present in the resulting automaton. Should we want to process a grammar set of substantial size and then store it in a memory, this could pose a problem, where we would end up with lots of identical structures in the memory. With our process, we are able to transform those structures into one unified supercombinator form. For example, three structurally identical branches of the automaton from Fig. 1 would yield supercombinators[1] showed in (2) and (3), where L^0 represents identity combinator. Terminal symbols from the original grammars are now stored separately, although they are still connected to the supercombinators by references. Should we apply our supercombinators on arguments, we would obtain the original grammar structure.

$$L^2 = \lambda x_1.\lambda x_2.L^0\ x_1 + L^1\ x_2 \tag{2}$$

$$L^1 = \lambda x_1.\lambda x_1.(L^0\ x_1)^* \tag{3}$$

Although the algorithm presented in this paper is capable of processing virtually any form of a CFG, we are leaning towards using it for the natural language processing. Formal grammars describing formal languages tend to be rather short and therefore it is not so relevant to process them further. But with the acquisition of a large enough grammar set we can actually see valid results. This of course does not mean, that our process is restricted to the natural languages only.

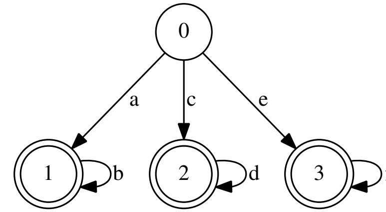

Fig. 1. Finite state automaton of a regular expression $a(b)^* \,|\, c(d)^* \,|\, e(f)^*$.

III. Transformation of Context-free Grammars into a Supercombinator Set

Our process transforms CFGs into a non-redundant supercombinator set. We have CFGs in the extended Backus-Naur form (EBNF) on the process' input. EBNF consists of rules and a set of terminal and nonterminal symbols. Rules are composed of symbols and grammar operations. In EBNF case, these operations are concatenation, alternative, closure and option. Our algorithm can process any number of defined operations, as they can be abstracted away the same way as the terminal symbols are. Every operation that occurs in the input grammar form is translated to our supercombinator form, yet the meaning of the operation remains the same. In the following example we use only concatenation and alternative operations for the simplicity sake.

Let's have a grammar defined by rules (4) and (5).

$$A \rightarrow a\ B\ a \tag{4}$$

$$B \rightarrow b \mid A \tag{5}$$

Rules (4) and (5) represent a simple CFG. We can see that this grammar generates language $a^n b a^n$. It contains a cycle, which will spice the things a bit. It also has only two rules, which in the resulting form would not show the full benefits of our process as there are no reoccurring structures. However, it is sufficient for this explanation.

The rule (4) is a plain sequence of three symbols. Rule (5) on the other hand is an alternative. Both rules refer to each other. This simple grammar can be transformed into a set that has three supercombinators, see Table I.

Supercombinators are lambda expressions. We use enriched lambda calculus, where the standard definition of lambda calculus has been enriched with grammar operations of the processed grammar. Hence in our example, only alternative and concatenation are added to it[2]. Supercombinators created from the grammar in (4) and (5) are shown in the Table I. Supercombinators are designated with the L symbol. Grammar operations are designated with the standard | symbol for alternative and + symbol for concatenation[3].

[1]Note the inclusion of grammar operations in the lambda expressions, see section III for further details.

[2]The example in the section II uses concatenation and closure.

[3]We have chosen the plus symbol since standard symbols for concatenation, either dot (.) or an empty space already are used in the lambda calculus.

TABLE I
SUPERCOMBINATOR FORM OF THE GRAMMAR IN (4) AND (5).

Name	Supercombinator Body	Arguments
L^0	$\lambda x_1 . x_1$	$\{a, b\}$
L^A	$\lambda x_1 . \lambda x_2 .\ L^0\ x_1\ +\ L^B\ x_2\ x_1\ +\ L^0\ x_1$	$\{a\, b\}$
L^B	$\lambda x_1 . \lambda x_2 .\ L^0\ x_1 \mid L^A\ x_2\ x_1$	$\{b\, a\}$

The arguments on the right side of the Table I are a part of our set. They represent permissible arguments for each supercombinator. They are stored non-redundantly as well. They are connected to the starting (top) supercombinator that roughly corresponds to the starting nonterminal symbol of a grammar. Should we apply those arguments to that supercombinator, we obtain the original grammar back. Therefore our supercombinator form is equivalent to the original CFG.

In the Table I we see that the grammar represented by rules (4) and (5) has been transformed into three supercombinators. As mentioned above, the fact, that we have obtained three supercombinators from two rules is due to the simplicity of the input grammar. What is important however, is the fact that our form is non-redundant. When we have larger grammars with repeating structures, our process works rather well, as we show later on in the section IV. Let's just imagine that we have expanded the grammar in (4) and (5) with same structured rules that have different terminal symbols. In that case, our form would not contain any new supercombinators, since it looks only at the structure and abstracts terminals away. Only new terminals and their links would be the new additions to our form.

A. Construction of Node Graph

The first version of our algorithm [10] could process only regular grammars. Construction of the form was straightforward and rather simple. In [11] we showed a method how to extend this process to CFGs. There we note that nonterminals could be treated similarly as terminals. Nonterminal in a rule's body means a jump to another nonterminal rule. And as we see on the grammar in (4) and (5), those jumps can create cycles.

Each rule produces its own subset of supercombinators. They are merged together later on, but at this stage they are treated as separate entities. On a plus side, this opens a possibility for a parallel processing. Since each subset of supercombinators contains a top supercombinator of that rule, i.e. a supercombinator that corresponds to the nonterminal from the left side of a rule, each subsequent call of another supercombinator inside of a rule's body can be replaced by that particular subset's top supercombinator. We just need to know the possible arguments and therefore the arity of that supercombinator.

In order to obtain that information, we need a graph constructed from the entire grammar. We need to know how many arguments are permissible for each nonterminal symbol. For that we are going to use depth first search from that node. To better show what we mean, let's have this following grammar (6):

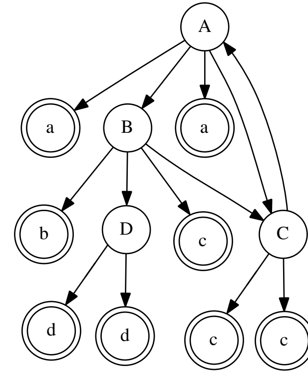

Fig. 2. The graph constructed from the grammar in (6).

$$
\begin{aligned}
A &\rightarrow a\ B\ a\ C \\
B &\rightarrow b\ D\ c\ C \\
C &\rightarrow c\ c\ A \\
D &\rightarrow d\ d
\end{aligned}
\tag{6}
$$

In this case we have four nonterminal symbols A, B, C and D and four terminals a, b, c and d. The graph of this grammar is depicted in Fig. 2. By using depth first algorithm from the node A, we obtain the following string: $A\, a\, B\, b\, D\, d\, d\, c\, C\, c\, c\, a$. By deleting nonterminals and removing all duplicates, we obtain the resulting argument string $abdc$. Now we can create a dummy version of a top supercombinator from the rule A. We know that it has four arguments and that is sufficient for us to use it inside of another supercombinator's body.

But do we need to remove duplicate terminals from that string? It seems that it is a logical step to retain the non-redundancy property. Yet as we show further on (see section III-D), this may not be the case. By not removing duplicates in this step we can obtain larger amount of similar structures, where the only trade off is an increased amount of connections to arguments.

B. Initial Supercombinator Construction

As all of our references to other rules are taken care of, we can now process each rule separately into a supercombinator form. Usually this yields at least two supercombinators per rule, the identity function supercombinator that we call L^0 and a top supercombinator of that rule. In case that the rule is more structured, other supercombinators are created. The amount depends on the structural complexity of each grammar rule. Each grammar operation inside of a rule creates its own supercombinator. As mentioned above, each nonterminal is replaced by the reference to its top supercombinator, which does not need to be created yet. This allows us to process rules in any order, even in parallel. After this step, we can proceed to the merge process that merges all structurally identical supercombinators together. This is done on the level of a single

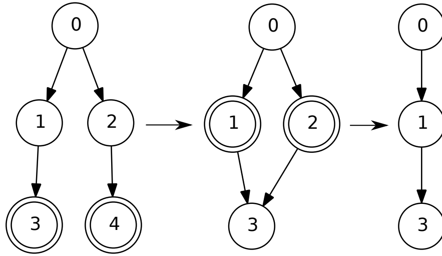

Fig. 3. Visualization of iterative merge operation.

rule and then over the entire set, thus creating unified non-redundant form.

Let us explain the merge process in more detail here. In Fig. 3 we see an example of supercombinator applications. On the left side we see a tree structure, where bottom nodes are supercombinators that are a part of supercombinators above them. This means that supercombinator with the identifier 0 contains in its body supercombinators 1 and 2. And they contain each only one subsequent supercombinator (3 and 4) in their bodies. We find out at the beginning of our merge process that nodes 3 and 4 are identical. We merge them together to a single node, designated as 3. Even if the nodes 1 and 2 are structurally identical, they are not merged yet, since they both contain different references. After the first merge iteration, we update references and now nodes 1 and 2 can be merged together assuming they are identical. We show the result on the right side of the Fig. 3. The set now contains only three supercombinators out of the original five. Needles to say, supercombinators that are different in the structure are not merged together, as they represent separate structures. The merge process stops when no new identical supercombinators are found after the reference update.

C. Transformation to a Single Set

We already have all basic functions to create a unified set of supercombinators. After processing each rule separately, we may merge them together with the same process that we have used before. After that we have a non-redundant set.

However this set is not necessarily final, we can still add another processed grammar into it, hence we achieve scalability. Imagine that we have processed a grammar G_1 to a set S_1. A new grammar G_2 appeared on the input. First we need to process that grammar into its own supercombinator set S_2 and than merge it with the S_1. Unification of grammars can be described as the following expression $G_1 + G_2 = S_1 \cup S_2$. Therefore we can continuously add grammars to a single set.

The ability to incrementally expand the set is an important property of our process. We can thus create one set of supercombinators from multiple grammars. We show how our set grows with the addition of new grammars in the section IV.

As already mentioned before, our algorithm is capable of transforming any CFG into a set of supercombinators. CFG is a formalism that can describe languages with certain properties. All rules of this grammar type are basically derivations of nonterminal symbols, as the basic structure of a CFG rule is $A \rightarrow \alpha$, where A is nonterminal symbol and α is a sequence of terminal and nonterminal symbols. We see that a sequence is an operation, which is transformed along with its arguments into a supercombinator. Each grammar operation creates exactly one supercombinator in this step. We can therefore say that any type of a rule can be transformed into a supercombinator. However, we obtain better results when we use restricted forms of CFGs as there is a higher chance that the repeating structures will occur. For example cycles in a grammar are quite restricting and limit that occurrence to a certain degree, as we can see on an example presented in the Table I.

D. Room For Improvement

The results presented in this paper are achieved with the use of our algorithm that has been improved and now differs in some points from the one presented in [4]. We have unified the merge operation and also used more efficient data structures that increased the speed of grammar processing. In the previous version, we have differentiated between the merger of rules within a single grammar and the merge in between the grammars. As we ultimately are getting a single set, the differentiation was unnecessary and now we are using the same merge operation during the entire transformation process.

As we have mentioned a bit earlier (see III-A), one of another improvements is the possibility of not deleting duplicate arguments from a string obtained from the grammar's graph. One of our process' core principles is the fact that we do not store the same element twice in the final set and not deleting duplicates might pose the risk of introducing redundancy. However, we argue that this is not the case and it can even reduce the amount of supercombinators, where only the number of connections would rise.

Let's have supercombinators (7) and (8):

$$L^A = \lambda x_1.\lambda x_2.x_1 + x_2 \tag{7}$$

$$L^B = \lambda x_1.x_1 + x_1 \tag{8}$$

We see that the first one has two distinct arguments, where the second has only one. Yet the structure of their bodies is suspiciously similar. It is just a simple concatenation of two arguments. What we see is that if we delete duplicate arguments, as it is in the case of supercombinator (8), we obtain supercombinators that are structurally similar yet different in their arity.

Should we treat all arguments as unique entities, the supercombinator (8) would not exist, as it would be merged with the (7), see the right side of Fig 4. We see that by treating arguments as unique elements, we actually obtain more compact form. At least here it is clear only in a theory. We have performed experiments to confirm this hypothesis, see section IV-B.

But does the inclusion of all arguments violate our non-redundancy criteria? No, it does not, since the arguments are stored separately, only the connections (in fact references)

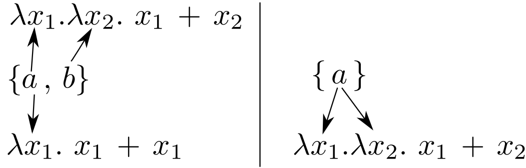

Fig. 4. Visualization of our old and new approach to argument connections.

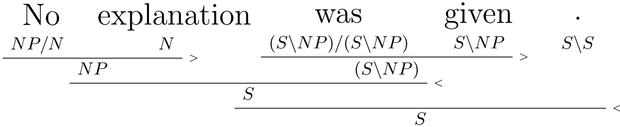

Fig. 5. Example of input sample in CCG form.

are actually attached to supercombinators. We see this fact in Fig. 4, where we see that this improved approach does not store anything more than once.

IV. Experimental Results

We present various experiments that show the abilities and properties of our algorithm in this section. As already mentioned, we are no longer using grammars generated with the Sequitur algorithm as we have done in [4]. We have decided to use different kind of input data.

In order to properly examine our algorithm, we need to have a rather large dataset. We are taking our input grammars from Groningen Meaning Bank (GMB) [6]. It is a large base of news articles parsed with Combinatory Categorial Grammar (CCG) [12]. At the time of writing this paper, GMB consisted of 10 000 short newspaper articles, 62 008 sentences in total. We want to have a set of structural data that is sufficiently large enough, and this bank matches that criteria. Normally, CCGs are used for parsing natural language sentences along with their semantics. However, we do not use those grammars in a traditional way, since they are already parsed and combined with deep semantics. We transform the tree structure of each parsed sentence into one CFG in a straightforward fashion. As an example, we can see input CCG in Fig. 5. It is a parse tree of a sentence "No explanation was given". This is the third sentence from the GMB sample no. 88/0248. It is short enough to serve as an example. The resulting CFG from that sentence is shown in Fig. 6.

To complete the picture, we show in the Table II supercombinators that are created from this sample. Arguments were omitted for brevity. You can see that rules 2 and 3

$$\begin{aligned} 0 &\rightarrow \quad 1 < . > \\ 1 &\rightarrow \quad 2 \;\; 3 \\ 2 &\rightarrow \quad < No > < explanation > \\ 3 &\rightarrow \quad < was > < given > \end{aligned}$$

Fig. 6. Context free grammar created from input sample.

TABLE II
Supercombinators created from the grammar shown in Fig. 6.

Name	Supercombinator Body
L^0	$\lambda x_1.x_1$
L^1	$\lambda x_1.\lambda x_2.\ L^0\ x_1\ +\ L^0\ x_2$
L^2	$\lambda x_1.\lambda x_2.\lambda x_3.\lambda x_4.L^1\ x_1\ x_2 + L^1\ x_3\ x_4$
L^{top}	$\lambda x_1.\lambda x_2.\lambda x_3.\lambda x_4.\lambda x_5.L^2\ x_1\ x_2\ x_3\ x_4 + L^0\ x_5$

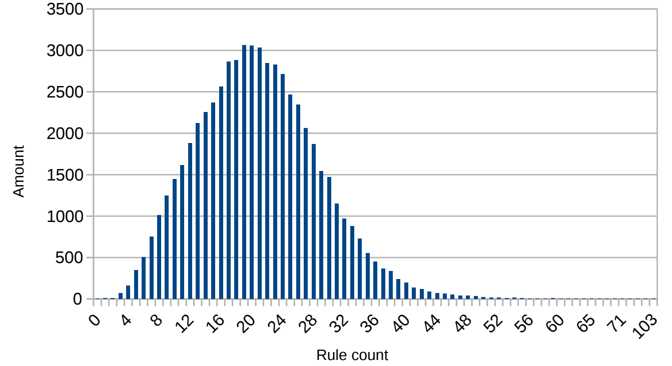

Fig. 7. Input grammars' rule amount distribution.

from Fig. 6 are structurally identical and they translate to the supercombinator L^1. Should we perform β-reduction of L^{top} with the arguments (in this case words), we would obtain the input sentence. Therefore our form is complete and fully represents the input sentence.

These grammars are different from the Sequitur grammars that we have used before. They are still simple CFGs that generate sentences and use only sequencing. However, and this is important to stress out, these CFGs are no longer just compiled from repeating phrases, but are purposely parsed based on their linguistic categories. We capture the structure of these parse trees, which in it self might show interesting information about the input form.

A. Input Data

We have obtained large amount of data by using GMB data transformed to CFGs. These data can show, how our algorithm works and what are its strong parts. We have transformed a total number of 62 008 sentences to the equal number of CFGs. The average number of rules per grammar is 20.838, with the median of 20 and the mode is 19. Standard deviation of rule amount is 7.947. We can see in Fig. 7 that the distribution of grammar rules roughly resembles the gauss curve, therefore it can be considered a normal distribution.

We have mentioned earlier in the section III-B that each grammar is processed into its own supercombinator set. How we can relate CFGs with the supercombinators created from them? We could look at the maximum arity of a supercombinator set, i.e. the arity of the top supercombinator. We see in the Fig. 8 that the distribution of maximum arity is also normal and very similar (although not identical) to the distribution of grammar rules. The other values are similar as well, where

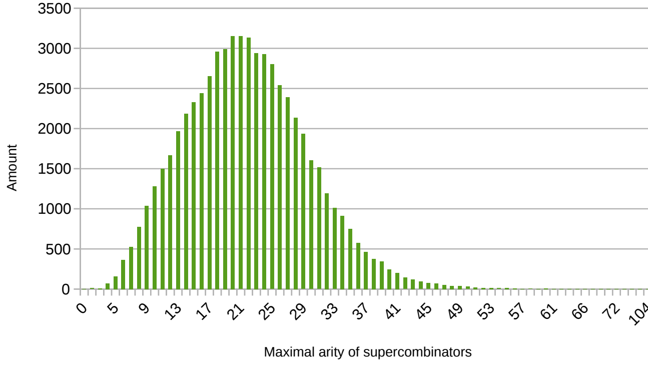

Fig. 8. Maximum arity of a supercombinator per supercombinator set created from input samples. Each input grammar generates one set, they were not merged yet at this stage.

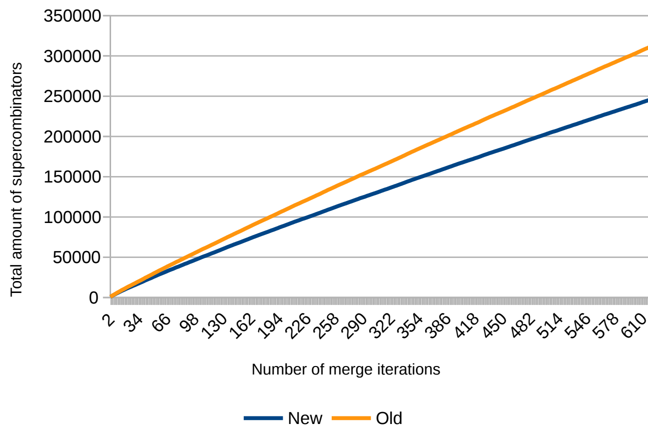

Fig. 9. Comparison of cumulative incremental merge between older and newer approach, described in the section III-D.

the average arity is 21.84, with the median of 21 and the mode of 20. Standard deviation is almost identical, 7.95. The arity is important property of supercombinators, as with it we can find out the theoretical maximum amount of created supercombinators.

B. Comparison of Approaches

We have described in the section III-D our performed tweaks to the algorithm. As this is the result section, we present the comparison results here. We have said that by allowing the same argument to be applied in one supercombinator more than once, we gain the reduction of elements.

To check our hypothesis, we have performed following experiment. We have taken the entire sample set and incrementally built a supercombinator set. This means that we have started with the first sample, created supercombinator set from it and then incrementally merged each next sample's set with it (see section III-C). In Fig. 9 we see that the growth of supercombinator amount in the form is lower in the case of our improved method. So these results confirm our hypothesis that the resulting form would contain fewer supercombinators.

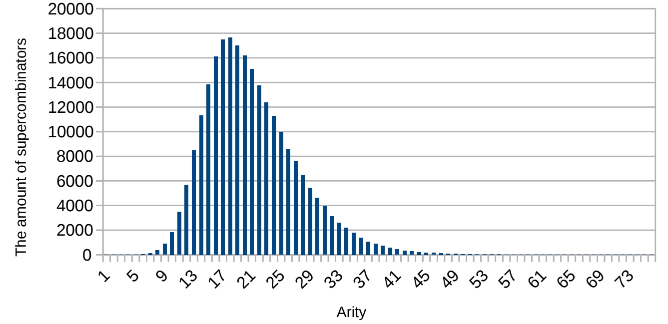

Fig. 10. The amount of supercombinators in the final form divided by their arity.

TABLE III
AMOUNT OF SUPERCOMBINATORS SEPARATED BY THEIR ARITY.

Arity	Amount	Catalan no.	Arity	Amount	Catalan no.
1	1	1	9	878	1430
2	1	1	10	1836	4862
3	2	2	11	3474	16796
4	5	5	12	5686	58786
5	14	14	13	8470	208012
6	42	42	14	11328	742900
7	128	132	15	13859	2674440
8	360	429	16	16099	9694845

C. Scalability of the Supercombinator Form

You may notice from the Fig. 9 that the growth of our form seems to be linear. That is because the maximum arity of our input samples is rather high. In the Fig. 10 we show the amount of created supercombinators in the final set split by their arity.

The intuition tells us that the amount of supercombinators that can be created for each arity is limited. If we have a non-redundant form, there must be some amount that cannot be surpassed. The limit of theoretically possible supercombinators created from CFGs with binary rules is known as the Catalan number (9).

$$C(n) = \prod_{k=2}^{n} \frac{n+k}{k} \tag{9}$$

This fact makes sense, since we are using binary CFG rules, and one of the counting problems that Catalan number describes is the number of successive applications of a binary operator. Should we split our form by the arity, we see in Fig. 11 that in each case the total amount never surpasses the Catalan number. Note that 0th Catalan number equals to our arity of 1. As Catalan number grows exponentially, we use exponential y-axis in Fig. 11. Even with it, the Catalan number (red line) rises steeply, quickly surpassing the amount of created supercombinators of higher arities.

In Table. III[4] we see the amount of supercombinators taken

[4]We do not show the entire set for brevity.

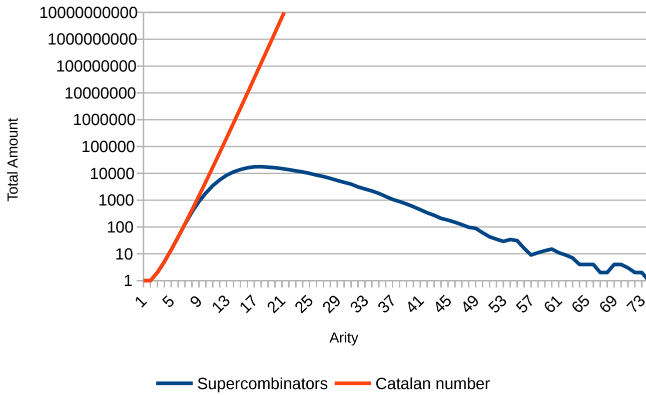

Fig. 11. Arity split supercombinator form with a logarithmic scale along with the Catalan number limit.

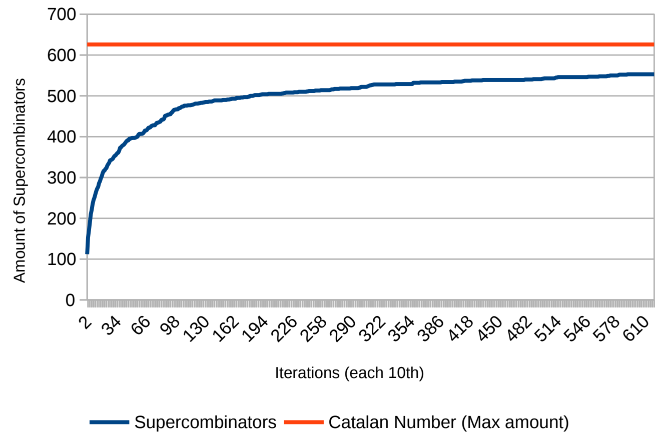

Fig. 12. Cumulative sum of supercombinators constrained with the arity of at most 8, along with the maximum limit, which is sum of the first 8 values of Catalan number.

from our set split by arity up to the number of 16. We obtain all theoretically possible supercombinators up to the arity of 6. Then we see that the amount of supercombinators with larger arities is orders lower than their corresponding Catalan number.

Although our set shows linear growth (blue line in Fig. 9), should we restrict the supercombinator creation process to some arity, we should see logarithmic growth. Arity restriction means that we do not allow any supercombinator with higher arity to enter the final set. As we use grammars that do not contain cycles, supercombinators cannot contain inside of them any higher or equal arity supercombinators, therefore our limiting does not require any special attention.

To demonstrate this limit, we have chosen to restrict arities starting above the number 8. This number has been chosen since it produces sizable amount of supercombinators yet the Catalan number for it is not that much higher, as it is for higher arities, see Table III. In Fig. 12 we see that our growth is now logarithmic, it does not surpass the limit imposed by the sum[5] of the first 8 numbers of the Catalan number (red line).

D. Identification of the Reoccurring Structures

Each supercombinator represents some part of a grammar structure. It is reasonable to assume that some structures do occur more often than others. That information is not available in our final set, since the set is not redundant. However, we can capture that information during the merge operation.

When we perform incremental growth of our form, we merge a supercombinator set created from one grammar with the rest of already created supercombinators. Therefore, we just need to count how many times has any supercombinator been merged. That would give us another parameter to our supercombinator form, the merge rate. In other words a number designating how many times has a certain supercombinator tried to enter the output set.

[5]We need to sum the first 8 numbers of Calatan number sequence, as the set now contains supercombinators with arities of the range from 1 to 8.

TABLE IV
FREQUENCY OF OCCURRENCE OF STRUCTURES.

Frequency interval	Amount	% After merge	% Before merge
≥ 1000	59	0.024	47.07
$< 1000, \geq 500$	43	0.018	3.38
$< 500, \geq 100$	317	0.129	7.33
$< 100, \geq 50$	348	0.141	2.77
$< 50, \geq 10$	2717	1.103	6.08
$< 10, \geq 5$	3794	1.54	2.78
$< 5, \geq 2$	19142	7.77	5.38
$=1$	219894	89.252	25.2

With this information, we can find out the most reoccurring structures inside the input data. We have split our final supercombinator set by the merge rate, see Table. IV. For better representation, we have split the set to intervals, so it could be more evident how many times has a unique supercombinator tried to enter the final set. In the second to last column that represents actual percentage of supercombinators in the final set after merge operation, we see that the majority of supercombinators are unique in the first place, as 89% have never been merged. Around 7.77% of supercombinators have been merged only once. Therefore we can conclude that only a fraction of supercombinators present in the final form occur frequently as structures in input grammars.

Now let's focus on that fraction. In the last column of the Table IV we see the actual rate of occurrence before merge operation. It is no wonder that the supercombinators that have been merged more than $1\,000$ times have more than 47% of the share. These are the most occurring structures in the input set after all. In the Table V we show ten supercombinators that have been merged the most times. The number in the first (and fourth) row means its order in sequence and next to it is its composition, therefore $(0, 1)$ means that this supercombinator is composed of identity L^0 supercombinator and the supercombinator in that table with the rank of 1,

TABLE V
TEN MOST MERGED SUPERCOMBINATORS.

Rank	Arity	Merged	Rank	Arity	Merged
1 (0,0)	2	62008	6 (0,4)	6	16025
2 (0,1)	3	60114	7 (1,1)	4	14602
3 (0,2)	4	48851	8 (2,3)	5	12651
4 (0,3)	5	30343	9 (0,8)	5	8433
5 (1,0)	3	17028	10 (0,6)	7	7995

which is $L^1 = \lambda x_1.\lambda x_2.L^0\, x_1 + L^0\, x_2$[6]. We see that L^1 exists in each set created from input grammars, as its merge rate equals the amount of grammars processed. Should we look at it from the CCG perspective, it represents a basic application of two elements. L^0 also exists in all input grammars sets, yet it is not present in the table due to the implementation simplifications. It has the arity of 1 and it is always present in every supercombinator that we create.

Note that the supercombinator L^1 has its rate of occurrence equal to the amount of grammars. We have not been counting how many times has this supercombinator been created while creating a single set from one grammar, we are only counting merge rate when merging already created supercombinator sets (created from input grammars) with the final set. This might be a threat to validity of this results, but we argue that even in the current state our process is able to identify the most reoccurring structures, as we are identifying structures in between the input CCG trees.

The second most occurring supercombinator has the arity of 3. It does not occur in all grammars however, as its merge rate is $60\,114$. This is possible due to the fact that there exists another supercombinator with the arity of 3 that has the occurrence of $17\,028$. Both of those supercombinators might be present in a single input set, since their arity is rather low. Yet we see that the first one occurs around three times more often than the second one.

We can say that we have found a way to identify the most reoccurring structures in our input samples. Each supercombinator directly translates to the structure. For better explanation, we present the Fig. 13 that contains two most occurring structures with the arity of 8. There are 429 possible permutations of supercombinators with the arity of 8, out of which only 360 exist in our set. Supercombinators in Fig 13 are the most occurring with that arity. We see that these structures are plain binary trees, the black nodes mark the spot where the arguments enter our supercombinator form, i.e. they represent the L^0 supercombinator. Next to each tree is their merge rate. We see that the most occurring structure with the arity of 8 is the deepest possible tree for that amount of leaf nodes. Tree structure like that can represent a list structure. The second most occurring supercombinator with the arity of 8 contains the same structure with the arity of 5, and it contains above mentioned supercombinator L^1 twice.

[6]The composition of L^1 is therefore $(0, 0)$.

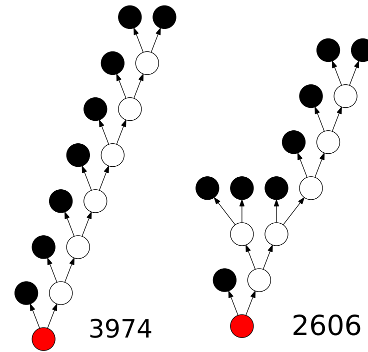

Fig. 13. Two most occurring supercombinators with the arity of 8.

V. DISCUSSION

We have presented large scale experiments on CFGs that have been taken from the structure of CCGs samples from GMB corpora. This is in contrast to our previous work [4] where we have used generated Sequitur grammars. These grammars are representable by trees, like our current samples, but Sequitur CFG tree nodes can have zero-to-n subtrees, where the trees of CFGs presented in this paper are binary. This means that current grammars tend to be narrower but they are also deeper. The largest supercombinator is represented by a tree with the depth of 22. It has the arity of 104, which means that it has 104 leaf nodes. It represents the longest sentence in the original GMB source.

Restriction to binary trees has allowed us to pinpoint the theoretical limit of our resulting form. In case of binary trees it is the Catalan number, and as we show, the amount of created supercombinators never surpasses the maximum possible amount set by this mathematical sequence. In fact, only supercombinators with the arity up to 6 are fully present in our final set. This result has been expected. A single natural language sentence can be parsed by CCG in multiple ways, caused by what is known as spurious ambiguity. Yet the final selected tree form is usually the most simple one. Therefore it is logical to expect that we won't have all possible supercombinators in our final form. Yet we have found out that a fraction, specifically 0.024% of supercombinators in the final form represents 47.04% of all structures in the input grammars that we have processed. This shows that natural language parsed with CCG tends to create similar structures rather than to create arbitrary ones. This conclusion is in order with the CCG spurious ambiguity property mentioned above.

The results presented in the section IV-D show that we can find the most used structures in the entire input set. This might be a little contribution to the filed of grammar metrics as we can now measure, observe and locate the substructures of grammars. However, the purpose of this paper is not a creation of a new metric. This might be the topic of our future research.

The results from section IV-B show that we might obtain smaller final forms, should we allow the identical arguments to be treated individually. This reduces the amount of supercombinators, as we reuse already created supercombinators. The

downside of this approach is larger amount of connections between arguments and supercombinators. The main feature of our form, the non-redundancy, still stands, as no two supercombinators in the resulting set are equal.

VI. Related Work

As our algorithm processes grammars, our work relates with the filed of grammar inference. Inference methods can transform linear text into a grammar form that we can further process and convert into non-redundant supercombinator set. There exist various methods of CFG (or their subset) inference, even from the positive samples only. Although due to Gold's theorem, it is not possible to infer grammar from positive samples purely algorithmically. Hrnčič, Mernik, Bryant and Javed used evolution algorithms [13] to circumvent that problem. Another methods might include the use of minimal adequate teacher, as used by Clark [14] or a rule based system presented by Dubey, Jalote and Aggarwal [15]. There are other methods to achieve that, De Higuera presents extensive survey of various grammar inference methods in [16]. Stevenson and Cordy in [9] describe methods of inference used in the software engineering.

As our results from the section IV-D indicate, our work might contribute to grammar metrics field. Grammar metrics are formal measurements of a grammar quality. Power and Malloy in [17] describe metrics and they split them into two types, size metrics and structure metrics. Črepinšek et al. build upon that and in [18] propose new metrics based on LR parsing. However, deriving new grammar metrics is not the purpose of this paper and would require additional research. But we believe that we might contribute this field in the future.

The algorithm from our work might be used to store grammars that are processed in a data-flow manner. This relates to the field of conceptualization [19], as supercombinators might represent concepts. Data obtained by such a process [20] can be transported into grammar forms and and then processed with our algorithm. As our research is grammar based, it might also prove useful to the Domain specific language (DSL) field [21]. DSLs are useful small languages that work with the abstraction rather well. They are primarily targeted for human-computer communication [22], and the structure containing data non-redundantly might prove to be useful.

VII. Conclusion

We have presented an improved version of supercombinator set acquisition algorithm in this paper. As in our previous version, this algorithm is capable to convert any CFG into a set of supercombinators accompanied with the arguments (terminal symbols). By application of arguments we obtain the input grammar back. The improvements presented here include non-removal of identical terminals in the creation of a supercombinator, using more efficient data structures and the unification of a merge process. Keeping the identical terminal symbols results into more compact form with less amount of supercombinators. Only drawback is the increased number of argument references.

We have performed experiments on a set of $62\,008$ sentences, taken from $10\,000$ news articles that are included in the Groningen Meaning Bank. The goal of our experiments was to prove that the supercombinator set is upper bound. We have found that in the case of binary CFGs, the limit is a mathematical sequence called Catalan number. Should we limit the supercombinators entering the final set by a relatively low arity (we have presented results limited with the arity of 8), the growth of that set is logarithmic and never surpasses the limit posed by the Catalan number.

Another experiments showed that our process can identify most reoccurring structures in input grammars. This might be a contribution to the field of grammar metrics. The results presented here show that supercombinator set obtained from natural language sentences contains only a small fraction of supercombinators that represent majority of structures in the input set, as our final set is non-redundant.

References

[1] R. J. M. Hughes, "Super-combinators a new implementation method for applicative languages," in *Proceedings of the 1982 ACM symposium on LISP and functional programming*. ACM, 1982. doi: 10.1145/800068.802129 pp. 1–10. [Online]. Available: http://dx.doi.org/10.1145/800068.802129

[2] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 331–380, Jul. 2005. doi: 10.1145/1072997.1073000. [Online]. Available: http://doi.acm.org/10.1145/1072997.1073000

[3] J. Kollár, M. Sičák, and M. Spišiak, "Towards machine mind evolution," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE, 2015. doi: 10.15439/2015F210 pp. 985–990. [Online]. Available: http://dx.doi.org/10.15439/2015F210

[4] M. Sičák and J. Kollár, "Supercombinator set construction from a context-free representation of text," in *Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on*. IEEE, 2016. doi: 10.15439/2016F334 pp. 503–512. [Online]. Available: http://dx.doi.org/10.15439/2016F334

[5] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical strcture in sequences: A linear-time algorithm," *J. Artif. Intell. Res.(JAIR)*, vol. 7, pp. 67–82, 1997. doi: 10.1613/jair.374. [Online]. Available: http://dx.doi.org/10.1613/jair.374

[6] V. Basile, J. Bos, K. Evang, and N. Venhuizen, "Developing a large semantically annotated corpus," in *LREC 2012, Eighth International Conference on Language Resources and Evaluation*, 2012. [Online]. Available: https://hal.inria.fr/hal-01389432

[7] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967. doi: 10.1016/S0019-9958(67)91165-5. [Online]. Available: http://dx.doi.org/10.1016/S0019-9958(67)91165-5

[8] L. Onnis, H. R. Waterfall, and S. Edelman, "Learn locally, act globally: Learning language from variation set cues," *Cognition*, vol. 109, no. 3, pp. 423–430, 2008. doi: 10.1016/j.cognition.2008.10.004. [Online]. Available: http://dx.doi.org/10.1016/j.cognition.2008.10.004

[9] A. Stevenson and J. R. Cordy, "Grammatical inference in software engineering: an overview of the state of the art," in *Software Language Engineering*. Springer, 2013, pp. 204–223. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36089-3_12

[10] J. Kollár, M. Spišiak, and M. Sičák, "Abstract language of the machine mind," *Acta Electrotechnica et Informatica*, vol. 15, no. 3, pp. 24–31, 2015. doi: 10.15546/aeei-2015-0025. [Online]. Available: http://dx.doi.org/10.15546/aeei-2015-0025

[11] M. Sičák, "Higher order regular expressions," in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on*. IEEE, 2015. doi: 10.1109/EMES.2015.7158427 pp. 1–4. [Online]. Available: http://dx.doi.org/10.1109/EMES.2015.7158427

[12] M. Steedman and J. Baldridge, "Combinatory categorial grammar," *Non-Transformational Syntax: Formal and Explicit Models of Grammar. Wiley-Blackwell*, 2011.

[13] D. Hrnčič, M. Mernik, B. R. Bryant, and F. Javed, "A memetic grammar inference algorithm for language learning," *Applied Soft Computing*, vol. 12, no. 3, pp. 1006–1020, 2012. doi: 10.1016/j.asoc.2011.11.024. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2011.11.024

[14] A. Clark, "Distributional learning of some context-free languages with a minimally adequate teacher," in *Grammatical Inference: Theoretical Results and Applications*. Springer, 2010, pp. 24–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15488-1_4

[15] A. Dubey, P. Jalote, and S. K. Aggarwal, "Learning context-free grammar rules from a set of program," *IET software*, vol. 2, no. 3, pp. 223–240, 2008. doi: 10.1049/iet-sen:20070061. [Online]. Available: http://dx.doi.org/10.1049/iet-sen:20070061

[16] C. De La Higuera, "A bibliographical study of grammatical inference," *Pattern recognition*, vol. 38, no. 9, pp. 1332–1348, 2005. doi: 10.1016/j.patcog.2005.01.003. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2005.01.003

[17] J. F. Power and B. A. Malloy, "A metrics suite for grammar-based software," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 6, pp. 405–426, 2004. doi: 10.1002/smr.293. [Online]. Available: https://doi.org/10.1002/smr.293

[18] M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, and G. Roussel, "On automata and language based grammar metrics," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 309–329, 2010. doi: 10.2298/CSIS1002309C. [Online]. Available: https://doi.org/10.2298/CSIS1002309C

[19] N. Carvalho, J. J. Almeida, M. J. Pereira, and P. Henriques, "Probabilistic synset based concept location," in *SLATe'12—Symposium on Languages, Applications and Technologies*. Alberto Simões and Ricardo Queirós and Daniela da Cruz, 2012. doi: 10198/7062 pp. 239–253. [Online]. Available: http://hdl.handle.net/10198/7062

[20] S. Ristić, S. Kordić, M. Čeliković, V. Dimitrieski, and I. Luković, "A model-driven approach to data structure conceptualization," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 5. IEEE, 2015. doi: 10.15439/2015F224 pp. 977–984. [Online]. Available: http://dx.doi.org/10.15439/2015F224

[21] D. Lakatos, J. Poruban, and M. Bacikova, "Declarative specification of references in dsls," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. IEEE, 2013, pp. 1527–1534.

[22] S. Chodarev, "Development of human-friendly notation for xml-based languages," in *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 8. IEEE, 2016. doi: 10.15439/2016F530 pp. 1565–1571. [Online]. Available: http://dx.doi.org/10.15439/2016F530