Proceedings of the Federated Conference on Computer Science and Information Systems pp. 503–512
DOI: 10.15439/2016F334
ACSIS, Vol. 8. ISSN 2300-5963

Supercombinator Set Construction from a Context-Free Representation of Text

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

Abstract—**Grammars might be used for various other aspects, than just to represent a language. Grammar inference is a large field which main goal is the construction of grammars from various sources. Written text might be analysed indirectly with the use of such inferred grammars. Grammars acquired from processed text might grow into large structures as the inference process could be continuous. We present a method to decompose and store grammars into a non-redundant set of lambda calculus supercombinators. Grammars decomposition is based on their structure and each distinct element is stored only once in such a structure. We present a method that can create such a set from any context-free grammar. To prove this and to show the possible applications in the field of natural language processing we present a case study performed on samples from two books. Those samples are the entire Book of Genesis from The King James Bible and the first 24 chapters of War and peace by Tolstoy. We obtain context-free grammars with the Sequitur algorithm and then we process them with our method. The results show significant decline in the number of grammar elements in all cases.**

I. Introduction

REPRESENTATION of extracted information from text is a question that correlates with cognitive science as researchers try to emulate the processes that runs in our head [1]. Natural languages might differ from their formal counterparts, but they can be described with the same formal theory as Chomsky [2] pointed out almost 60 years ago.

From a more pragmatic stand point of view, extracted information may be represented in a grammar form. This is usually the goal of the grammar inference (or grammar induction) field. As Gold in [3] stated, only superfinite grammars can be inferred from a text. By text Gold means a set of positive samples, i.e. samples that belong to the language that inferred grammar is representing. Yet recent advances in this field has shown, that although in a strict formal way Gold theorem still holds, by using heuristic, statistical or evolutionary methods, we are able to infer more complex grammars, like context-free grammars (CFG). De la Higuera presents a review of possible inference methods in [4]. And not only formal languages can be inferred. We are able to perform inference on natural languages as well, as Onnis, Waterfall and Edelman point out [5].

As we infer grammars of large quantities of text, we may indeed obtain large grammars. If those texts are similar, for example we are processing books written in the same language by the same author, then the resulting grammars are in a risk of having a large number of rules and lots of similar information stored separately. This phenomena is called structural explosion. Where structurally similar, but symbolically different rules of grammar are inferred and then stored each separately. In this paper we present a way how to represent any CFG grammar in a non-redundant form. This form is a single structure composed of lambda calculus supercombinators. In the section II we reason why such a form is useful and how it relates to the field of natural language processing.

This work was supported by project KEGA 031TUKE-4/2016 "Integrating software processes into the teaching of programming".

The main contributions of this paper are:

- We present the entire process of supercombinator set construction from any CFG grammar. The theoretical background is presented in the section III and the detailed description itself is in the section IV. It is a multi step process, each step is explained in a separate section accompanied with appropriate examples.
- Our approach has been tested on larger scale experiments that we present in the section V. We used Sequitur algorithm [6] to create context-free grammars from book samples and then we run our process on them. We present the results, that show significant reduction of grammar elements. This indicates the prevention of structural explosion.
- In the section VI we discuss the possibilities of text analysis that results from the usage of Sequitur algorithm and our supercombinator set acquisition process. We point out that even from the simple grammars, that are the result of Sequitur algorithm, we can extract useful structures contained in a natural language text.

II. Motivation

The basic idea behind this work stems from the engineering discipline of grammarware [7]. Simply put, we can use grammars for other purposes, than just for the representation of a language. With our approach we can decompose and store a CFG into a non-redundant form. In this section we explain, what good that process brings and how it relates to the field of natural language processing.

Why do we need to store a grammar inside a non-redundant structure? If we were given a small, predefined, static CFG, that we use only as a base for a parser for example, than our process could indeed be redundant itself. However, we can obtain a CFG from previously unknown or rather unprocessed text. And we might want to process lots of such texts for purposes like information extraction, language learning, text analysis, grammar inference or others. And consider that we would want to obtain one structure from all of those texts. If we were to keep those grammars stored in a basic CFG form, we could find ourselves buried under structurally same, yet semantically different rules. This is so especially if we are to process grammars that are large yet their rules are rather simple, as we show in the section V. Such rules might differ only in symbols but all grammar operations are the same. So instead of having a sequence of two different symbols stored for each pair of symbols separately, we can store one structural representation of two symbol sequence and link it with its respective symbols. Those symbols are also stored non-redundantly, so in case we have large terminal symbols (like words), having links to them is more memory efficient.

Our approach is based on the lambda calculus principle. The idea of the application and abstraction that is inherent to the lambda calculus offers a way to represent grammar rules that are separated from their terminal symbols. Therefore the entire structure is represented as one big set of supercombinators. The applications of those supercombinators on other supercombinators might be perceived as links. In that case we may view such a set as some network like structure. We explain this principle in further detail in [8] and [9], but also here, in the section III.

Now, how our work relates to the natural language processing? Well, the supercombinator form has few advantages. The non-redundancy has already been mentioned. The second advantage is that it describes and decomposes the structure of grammar rules. This for example opens the possibility to a form of text structure analysis. By searching for similar structures of text, combined with appropriate grammar inference mechanism, we might obtain information that may be used for various forms of analysis, like author or style identification. However, this possibility is out of scope of this paper.

Our process strongly relates to the field of grammar inference. Should we apply our process on a finite string of symbols, we would obtain just two supercombinators, out of which one would be long and would represent entire sequence of words. And that is not what we want, since we want to capture structure. So we need to start from a grammar form.

In this paper we are processing simple (in the structure, not in the size) CFGs obtained by Sequitur algorithm application (see section V) on a text written in a natural language, in our case the English language. We do not need to use Sequitur algorithm only. There are many ways to infer (or induce) a grammar from any text, see section VII. Therefore we can use any CFG grammar in our process.

III. Background

First of all, we need to explain how the entire process of supercombinator form acquisition works. The first version of this process has already been explained in [9]. There we have used only regular languages to create our supercombinator set. And thus the process was left in the theoretical space, i.e. we have not used real world case study. Our other work [8] presented an introduction to context-free languages application via higher order principle. In this section we build up on this principle towards building complete set of supercombinators from any CFG.

A. Enriched Lambda Calculus

Ordinary lambda expression e is defined by the rule (1).

$$e \rightarrow a \mid x \mid e\ e \mid \lambda x.e \tag{1}$$

Where a represent any constant, x a lambda variable, $e\ e$ is lambda application and $\lambda x.e$ is lambda abstraction. We can enrich this simple definition with grammar operations, thus the result of lambda expression reduction would not be a single value but the grammar expression, either regular or, as we later show, even context-free. As an example for regular-language-enriched lambda calculus have an expression (2).

$$L = \lambda x_1.\ \lambda x_2.\ x_1 \mid x_2 \tag{2}$$

This expression takes two variables as arguments and results into a regular expression, that consists of the alternative operation applied on both arguments. Therefore lambda application $L\ a\ b$, where a and b are terminal symbols, yields a regular expression $a \mid b$. We may notice, that we have used an infix notation for the alternative operation. This is only a syntactic sugar however.

To enrich our basic lambda calculus definition (2) we just add a regular expression option as another alternative. The formal definition of that regular expression is shown in (3).

$$r \rightarrow a \mid r_1 \mathbf{+} \ldots \mathbf{+} r_n \mid r_1 \boldsymbol{|} \ldots \boldsymbol{|} r_n \mid \boldsymbol{(}r\boldsymbol{)^*} \mid \boldsymbol{(}r\boldsymbol{)} \tag{3}$$

The first element represents terminal symbol. Then the structures of concatenation, alternative and Kleene closure meta-operations are defined. The final element represents ordinary bracketing. Note, that the operators of regular expression themselves are depicted in a bold font, so we can distinguish them from the meta-operators. Concatenation is usually depicted without its operator or with $'.'$ operator. We have used the $'+'$ operator as the dot is already used in the lambda abstraction. Also, the expression $r_1 + r_2$ is equal to the expression $r_1\, r_2$. Should we use operator-less notation in our extended lambda calculus notation however, it could cause a confusion between a lambda application and the concatenation itself. Therefore we are going to stick with the plus operator should we use the concatenation inside of a lambda expression.

The fact, that we may define lots of different and specialized operations for regular expressions is accounted for in this paper. We are not restricting our algorithm for supercombinator form construction with predefined static regular expression operations (i.e. only those three defined in (3)). We use an abstract function that acts as a placeholder for any operation. This idea is further explained in the section IV-A.

We show in Tab I a supercombinator set obtained from the expression $a\, b \,|\, (c)^*$. This expression serves as a good example, since it contains the alternative, concatenation and

TABLE I
SUPERCOMBINATOR SET FOR GRAMMAR $ab|(c)^*$.

Supercombinators			Arguments
L^0	=	$\lambda x_1.\ x_1$	$\{\ a,\ b,\ c\ \}$
L^1	=	$\lambda x_1.\ L^0\ x_1$	$\{\ a\ \}$
L^2	=	$\lambda x_1.\ \lambda x_2.\ L^1\ x_1 + L^0\ x_2$	$\{\ a\ b\ \}$
L^3	=	$\lambda x_1.\ (L^0\ x_1)^*$	$\{\ c\ \}$
L^4	=	$\lambda x_1.\ \lambda x_2.\ \lambda x_3.\ L^2\ x_1\ x_2\ \vert\ L^3\ x_3$	$\{\ a\ b\ c\ \}$

closure operations. The set itself was obtained with the use of process defined in [9]. The process in this paper is slightly different, but any actual difference is pointed out.

The important accompanying part of any supercombinator is its permissible argument string set. There may be more than one permissible argument string for each supercombinator as we can see in the case of L^0. Al three terminal symbols are possible arguments in this case. Only one argument string is allowed for any other supercombinator. We obtain the original expression $ab|(c)^*$ by β-reduction of the supercombinator L^4 with its argument string abc. The original expression would not be the result, if we allow any other argument strings to accompany that supercombinator. Note, that the permissible arguments are represented in the memory only once. They are connected with their supercombinators with the use of links. Argument strings are necessary for the reconstruction of original expression, any other argument string would lead to different expressions being created.

Supercombinators in Table I represent a structurally decomposed form of regular expression $ab|(c)^*$. Some of those supercombinators are applied more than once, like L^0. This is the solution to the structural explosion, since no multiple occurrences of equal supercombinators exist. There exists one top supercombinator for every expression It's the one by which β-reduction we obtain the original expression. In this case it's the L^4. The supercombinator form is reusable. So if we were to process more expressions, all already existing supercombinators would be reused. Only new arguments links would be added in that case. A simple example: if we were to add expression consisting of one symbol, say d, the supercombinators L^0 argument set would be enriched by the d symbol and at the same time it would become top supercombinator for that expression (but only if used with the d symbol).

Each supercombinator represents a part of the entire expression structure. As we see in Table I, the identity function is represented by L^0, a sequence of two variables by L^2 or a closure over one variable by L^3.

B. Higher Order Regular Expressions

We have published a paper under this name [10] where we noted that the difference between regular and context-free expressions is not really that significant. And that we may view context-free expressions as higher order regular expressions, i.e. expressions that may take another expression as an argument.

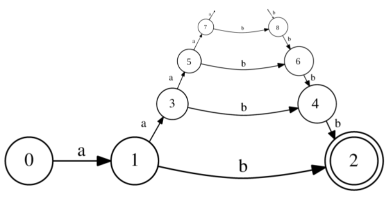

Fig. 1. The infinite state automaton obtained from higher order expression $A \to aAb \mid ab$.

The simple example is shown in the expression (4).

$$A \to aAb \mid ab \tag{4}$$

This expression represents the language $a^n b^n$, standard example of a nonregular language. The expression (4) is in BNF form, but we may view it as an expression, where we have an alternative of two subexpressions. The first one is a sequence of three symbols. The middle symbol represents a jump inside another iteration of expression evaluation. Expression 4 is represented by the automaton depicted in Fig. 1. It's an infinite state automaton. The important fact here is the idea that there is not that big of a gap between regular and context-free grammars.

This idea of higher order expression translates into our enriched lambda calculus easily. The higher order jump is nothing else but an ordinary lambda application. The first element in the alternative of expression (4) can be translated into the following lambda expression:

$$L^{aAb} = \lambda x_1.\lambda x_2.\ x_1\ +\ (L^A\ x_1\ x_2)\ +\ x_2 \tag{5}$$

Where L^A is the top supercombinator for the expression (4), from which the supercombinator L^{aAb} is being applied. In this scenario, both supercombinators have the same arguments, so there is no need to extend the possible argument set for supercombinators in the lower levels of hierarchy.

It's important to note, that supercombinator (5) is not in the final, complete form, since it applies directly its arguments to the result. In the final form, argument applications are always translated into the identity supercombinator $L^0 = \lambda x.\ x$.

C. Basic Idea

We have presented the idea of regular-language-enriched lambda calculus and the way to view CFGs as context-free expressions or higher order regular expressions. Now we are going to show the outline of our process, where we translate those expressions into a supercombinator set.

Every context-free expression is based on a set of context-free grammar rules. Under the BNF definition (or EBNF if were tu use closure and option operations), these rules have form of $A \to r$, where A designates the nonterminal and r an expression into which the nonterminal is transformed. The expression has a form as defined in (3) with the addition of nonterminal symbol as a possible element.

Any nonterminal symbol may have more rules that define its expansion. But those rules can be merged together with the alternative operation without any consequences. , process in (6). Our approach requires all such expressions to be merged together.

$$\left.\begin{aligned} A &\to ab \\ A &\to (c)^* \end{aligned}\right\} \Rightarrow\ A \to ab \mid (c)^* \tag{6}$$

There exist one top supercombinator for every expression. This is the supercombinator from which we are able to reconstruct the original expression by applying it to its permissible argument string. We may extract that string even before we start to transform the expressions. This is an important property for implementing higher order jumps.

Simple outline of our transformation process is:

1) First step is to transform the input into the internal expression tree form. Each rule is transformed into separate tree, thus we obtain a set of trees, precisely one tree per nonterminal. Each tree is named after the nonterminal, that it represents.
2) Then we obtain the list of permissible arguments for every tree in the set. These lists do not only include arguments obtained by searching the tree for terminal symbols, but also include all terminals inside each nonterminal accessible from the current expression tree. Each terminal symbol occurs in every final argument string precisely once. No duplicates are allowed.
3) Each expression tree is transformed into separate, independent set of supercombinators. Each supercombinator of that set has exactly one argument string, since they haven't been merged yet. Every occurrence of nonterminal in expression is replaced by temporary supercombinator that points to that nonterminal. We have already acquired their permissible argument strings in the previous step.
4) Equal supercombinators are merged for each expression separately. Supercombinators now may contain more than one argument string in their permissible argument set. This process is done for each expression independently, thus may be performed in parallel.
5) Temporary supercombinators representing other nonterminals are now replaced by top supercombinators of expression represented by those nonterminals.
6) Equal supercombinators from all expressions are new merged together. The result of this is a single supercombinator set that represents a structurally decomposed original grammar.

IV. Supercombinator Set Acquisition from Context-free Grammar

We present the details of the transformation process in this section. All the steps of transformation are illustrated with appropriate examples.

A. Tree Creation

We start our process in the similar fashion as in our previous work, by expression transformation into a tree form. However, the trees used in the current process are different.

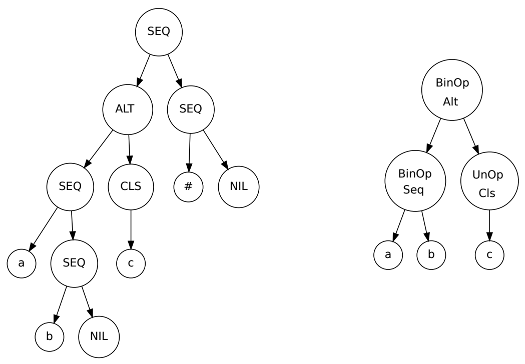

Fig. 2. The old and the new type of expression tree for $a\,b \mid (c)^*$.

Listing 1. Type of a context-free expression tree.

```
data Tree = Leaf Term | Jump NTerm
   | UnOp OpName Tree | BinOp OpName Tree Tree
   | MultiOp OpName [Tree]
```

As mentioned earlier, we tried to separate the operations from the basic structure. While in our previous work, the operations themselves were part of a tree as nodes, now they have been abstracted away.

We have selected $A \to a\,b \,|\, (c)^*$ as an example expression. Its depiction on Fig. 2 shows the old and the new type of tree. This figure immediately shows the difference in node amounts. The current tree form, the one on the right, is much more space efficient.

The formal definition of this new tree form is depicted in Listing 1. Constructors `Term` and `NTerm` represent terminals and nonterminals respectively. We can see that instead of a direct use of an operator as a node, we use abstract nodes. They are `UnOp` for unary operations, `BinOp` for binary operations and `MultiOp` for n-ary operations. `OpName` holds the name of that operation, which semantic is not defined here, since for now we only care about the structure. The semantics needs to be supplied for each intended operation for the lambda calculus to work, but it's presented independently from a tree.

We have implemented two different ways of processing operations. One is by constructing the tree only with the use of `MultiOp` nodes, that hold the entire subtree in a list. And the second is by using a combination of `BinOp` and `UnOp` nodes. This difference is crucial, since it produces different supercombinators, as we show in the section IV-C and further expose by the experiment in the section V-A. The list structure can hold any operation arity, therefore may be used exclusively. The other two operations should be used in tandem to achieve currying in application of supercombinators.

Currying is a well known phenomenon, where we may perceive any n-ary function as a sequence of unary high order functions that are applied to the arguments. We may view the sequence $abcd$ as either being transformed to the node $MultiOp(a, b, c, d)$ or as currying like tree $BinOp(a,\ BinOp(b,\ BinOp(c,\ d)))$.

Both tree approaches can be combined, as we did in our

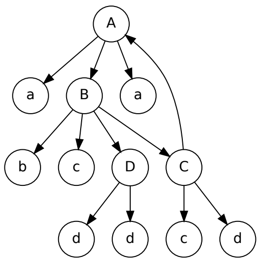

Fig. 3. Graph of elements from the context-free expression (7).

previous work. We have used `MultOp` like nodes for an alternative operation, `BinOp` type for a sequence and `UnOp` for a closure. We have also used special `Nil` node to sign the termination of a whole expression or a sequence. This special node is the reason, why there is L^1 supercombinator in Table I. However, it's equivalent to the L^0 supercombinator and therefore redundant.

The `Jump` node presents an actual jump to the nonterminal designated by the name in this node. This basically turns the set of trees into a directed, possibly cyclic, graph.

B. Argument Lists Extraction from Nonterminals

This step is essential for our process, since we are using jumps to other expressions. And each jump brings a new possibility of permissible terminal symbols at the input of their top lambda expression. Take expressions (7) as an example.

$$\begin{aligned} A &\rightarrow a\ B\ a \\ B &\rightarrow b\ c\ D \mid C \\ C &\rightarrow c\ d \mid A \\ D &\rightarrow d\ d \end{aligned} \tag{7}$$

The only terminal symbol actually inside of the expression A is the symbol a. But others, namely b, c and d are accessible. As they are accessible from the B and the C expressions. The expression D has only the symbol d as accessible.

From a set of terminals and nonrerminals of all expressions, a single directed cyclic graph is constructed. Each nonterminal forms a node which links to its body, and terminals represent leaves. The graph of expression (7) is depicted in Fig. 3.

Afterwards, a depth first search is performed for every nonterminal node in order to find all possible symbols from it. If we omit nonterminals from such a path, only a string of terminals remains. The final step is the removal of all duplicates, so the symbol strings contain each symbol at most once. This process is shown in (8) for nonterminal A. Thus symbol string for expression A is *abcd*, *bcda* for B and *cdab* for C. D has only one symbol, d, as a permissible symbol.

$$\begin{aligned} &A \Rightarrow A\ a\ B\ b\ c\ D\ d\ d\ C\ c\ d\ a \Rightarrow \\ &\qquad \Rightarrow a\ b\ c\ d\ d\ c\ d\ a \Rightarrow a\ b\ c\ d \end{aligned} \tag{8}$$

C. Construction of Supercombinator Sets

The process described in this section is performed on all expressions separately, therefore can be executed in parallel.

Listing 2. Definition of a supercombinator data type.

```
data Lambda = LeafLambda LambdaId
 | JumpLambda LambdaId Nterm
 | UnLambda LambdaId OpName SubLambda
 | BinLambda LambdaId OpName SubLambda SubLambda
 | MultiLambda LambdaId OpName [SubLambda]
```

Every single expression yields independent supercombinator set. We used to perform this process with the use of a direct tree transformation in our previous work. We have chosen to do it differently this time.

The basic idea behind obtaining supercombinators is that every operation node of a tree yields exactly one supercombinator. Therefore we have defined a data type for them, shown in Listing 2. All constructors are named after the nodes of a tree they represent. Each of them has its own *id.* All *id*s are unique, since we haven't performed merge yet. The *id* references allow us to use linear form instead of interconnected tree form, hence we may perceive them as pointers.

Each supercombinator has its own permissible argument string. We have separated that string from the underlying data type in order to keep its core aspects tidy and in order. It's functionally irrelevant whether the argument string is represented directly within the data type or as an accompanying list of arguments within a tuple.

The `LeafLambda` constructor represents already mentioned supercombinator $L^0 = \lambda x\ .\ x$, specified only by its *id.* Its argument string consists only from the original symbol of a `Leaf` node. `JumpLambda` is temporary supercombinator that serves as a place-holder for expression name and hold permissible argument strings of that expression. We have already obtained them during the previous step. Other three forms of supercombinators are functionally similar since they represent operations. `SubLambda` represents a supercombinator that is being applied to that operation. It holds only a pointer to the real supercombinator and references to arguments inside the argument string.

The direct mapping between supercombinators and our internal form is shown on the example (9). The ids are upper indices of the L symbol. The arity of a lambda expression is obtained from the length of its argument string. The lists in `SubLambda` tuple refer to the index of an argument inside the argument string. Such an argument string has the length of 2 in this case.

$$\begin{aligned} &\mathit{BinLambda}\ 1\ \mathit{Sum}\ (2,\ [0,1])\ (0,\ [0]) \iff \\ &\qquad \iff L^1 = \lambda x_0 . \lambda x_1 . L^2\ x_0\ x_1 \mid L^0\ x_0 \end{aligned} \tag{9}$$

The permissible argument string for any supercombinator is obtained as a merge of their children arguments. Therefore it is really important to perform step 2, as the jump supercombinators now have an entire argument string of an expression they are referring to.

Using `MultOp` nodes exclusively in a tree yields different supercombinators as an equivalent tree composed of `BinOp` and `UnOp` nodes. The former results into n-ary operation lambda expressions, where one operation is applied over multiple sub expressions. The latter method usually yields more supercombinators, but each has at most two sub expressions applications in its body, i.e. its underlying operation is at most binary.

D. Merge within Expression Supercombinator Set

The previous step may yield supercombinators that are equal, only their argument strings may be different. All such supercominators are merged, so no duplicates are present inside each set. This step, as the previous one, can be performed in parallel over all expression sets.

The equality of supercombinators, formally described in [9], means, that in order for supercombinators to be equal, they need to have the same arity and need to contain same `SubLambda` elements. In terms of definition in Listing 2 that means equality of all elements excluding the *id*. All equal supercombinators need to be merged. Even some supercombinators which contain references that initially lead to different supercombinators might be equal, since the sub-supercombinators may have been merged in the previous iteration of the merge step. Example of a merger is presented in Table II, where we see the initial and merged supercombinator set of the expression $ab \,|\, cd$.

We start the merge process with the identification of all equal supercombinators within a set. Then we group them together. We now have some groups of equal supercombinators and the rest of the set. Those groups can now be merged, even in parallel. Well, only their argument strings are merged into a single set and only one supercombinator comes out of this process. Its *id* now needs to be updated in the entire set, replacing the old, now unused *id*s. After this update, new equal supercombinators may be present, so we need to repeat this process until no new equal supercombinators are found. The process depicted in Table II had two merge iterations. Argument sets always consist of strings with the same length, since the arity of equal supercombinators needs to be the same. The resulting set of supercombinators is now duplicate free.

E. Removal of Jump Supercombinators and the Merge of All Sets

At this moment, we possess numerous sets of supercombinators that we need to connect together and then merge them to a single set. Each temporary jump supercombinator is now replaced by a top supercombinator of an expression it points to. It is important to keep the *id*s in between the sets from clashing, i.e. each set needs to have different *id*s. After this step, another merge is applied that joins all the sets to a single, duplicate free set.

For a simple grammar (10) we obtain the final supercombinator set in lambda calculus form, depicted in table III. If we consider the nonterminal A to be the starting symbol, then the top supercombinator is in our case L^2.

$$\begin{aligned} A &\to aB \mid a \\ B &\to b \mid A\, b \end{aligned} \tag{10}$$

TABLE III
SUPERCOMBINATOR SET OF GRAMMAR (10).

Supercombinators	Arguments
$L^0 = \lambda x_0.\ x_0$	$\{\{a\},\{b\}\}$
$L^1 = \lambda x_0.\ \lambda x_1.\ L^0\ x_0 + L^4\ x_1\ x_0$	$\{\{a\ b\}\}$
$L^2 = \lambda x_0.\ \lambda x_1.\ L^1\ x_0\ x_1 \mid L^0\ x_0$	$\{\{a\ b\}\}$
$L^3 = \lambda x_0.\ \lambda x_1.\ L^2\ x_0\ x_1 + L^0\ x_1$	$\{\{a\ b\}\}$
$L^4 = \lambda x_0.\ \lambda x_1.\ L^0\ x_0 \mid L^3\ x_1\ x_0$	$\{\{b\ a\}\}$

If we use a special form of β-reduction, we obtain the grammar (10) back. By special form we mean replacing all top supercombinators in the body of expression by their nonterminal representation. This is an important step, since it prevents the infinite loop. Already mentioned L^2 supercombinator is the top for A and the L^4 supercombinator is the top for B. But should we use in the β-reduction only on the top supercombinator of an entire grammar, the A, we get a transformed version of a grammar, as shown bellow in (11). We can conclude, that we may use the supercombinator form to transform a form of grammar.

$$A \to\ L^2\ a\ b \to^*\ a\ (b \mid A\ b\ ) \mid a \tag{11}$$

V. EXPERIMENTAL RESULTS

We have been using only abstract grammar symbols so far. Those abstract symbols, like a for terminals and A for nonterminals are useful for the explaining purposes, but they do not represent anything we can find in the real world directly. Thus we have performed experiments on the book samples. As mentioned before, we obtain a supercombinator set that represents the structure of elements. But we cannot just use our process over a fixed sequence of words, since that would result into one big supercombinator (and one L^0 supercombinator of course). We need a grammar first. And for that reason we have chosen Sequitur algorithm.

Sequitur algorithm, created by Nevill-Manning and Witten [6], creates context-free grammar from a linear sequence of discrete symbols. The words of English language in our case. The resulting grammar generates only the input text, therefore it's possible to use our supercombinators for text analysis of that text, since no other text is possible to generate either from the Sequitur grammar or our supercombinator set.

We use *the Book of Genesis* from the King James Bible and chapters from Leo Tolstoy novel *War and Peace*[1]. We have chosen these books because they are written in different styles and the former has many unknown authors while the latter was written by a single well known author, therefore may be less schematic and fragmented. We use various samples from these two books, all of which are listed in Table IV. We also list abbreviations, by which we are going to reference them, and we list the total word count of each section as well.

[1]The books were obtained from Project Gutenberg, located at http://www.gutenberg.org, where they are distributed under GNU Free Documentation license 1.2.

TABLE II
EXPRESSIONS $ab \mid cd$ MERGE OF DUPLICATE SUPERCOMBINATORS.

Supercombinators	Arguments	Merged Supercombinators
$L^0 = \lambda x_1 . x_1$	$\{a\}$	
$L^1 = \lambda x_1 . x_1$	$\{b\}$	
$L^2 = \lambda x_1 . x_1$	$\{c\}$	
$L^3 = \lambda x_1 . x_1$	$\{d\}$	$L^0 = \lambda x_1 . x_1$
$L^4 = \lambda x_1\, x_2 . L^0\, x_1 + L^1\, x_2$	$\{ab\}$	
$L^5 = \lambda x_1\, x_2 . L^2\, x_1 + L^3\, x_2$	$\{cd\}$	$L^1 = \lambda x_1\, x_2 . L^0\, x_1 + L^0\, x_2$
$L^6 = \lambda x_1\, x_2\, x_3\, x_4 . L^4\, x_1\, x_2 \mid L^5\, x_3\, x_4$	$\{abcd\}$	$L^2 = \lambda x_1\, x_2\, x_3\, x_4 . L^1\, x_1\, x_2 \mid L^1\, x_3\, x_4$

TABLE IV
LIST OF BOOKS AND THEIR SECTIONS USED IN THE EXPERIMENTS.

Book Name	Used Sections	Abbreviation	Words
Book of Genesis	Sections 1 - 3	*Gen3*	1429
Book of Genesis	Sections 1 - 6	*Gen6*	3840
Book of Genesis	Sections 1 - 12	*Gen12*	7306
Book of Genesis	All	*Gen*	39797
War and Peace	Chapter 1	*WP1*	2015
War and Peace	Chapter 2	*WP2*	1378
War and Peace	Chapter 3	*WP3*	1469
War and Peace	Chapter 4	*WP4*	1417
War and Peace	Chapters 1 - 4	*WP1-4*	6279
War and Peace	Chapters 1 - 12	*WP12*	17755
War and Peace	Chapters 1 - 24	*WP24*	37538

TABLE V
SEQUITUR GRAMMAR SAMPLE OBTAINED FROM THE BOOK OF GENESIS.

Rule		Rule body
78	→	`have dominion over the fish` **29** `sea` **79 96**
79	→	**56** `the`
80	→	`all` **61**
81	→	**56** `every`

The resulting Sequitur grammar uses only concatenation operation. Using Sequitur to decompose text of literature, i.e. text without rigorous structure, inevitably leads to the state, where the first rule consists of a long sequence of terminals and nonterminals, while the rest of the rules have rather low arity. Despite this fact, we still might obtain interesting results by performing our process, as the portion of lower arity rules might be represented by a single supercombinator. To imagine how Sequitur grammar looks, see Tab V, where we show a sample of the grammar obtained from *Gen*. We see that nonterminals are represented by numbers and terminals by actual words.

What do we expect from our experiments? Since our process captures the structure of rules, we expect the processed Sequitur grammar to have less elements, represented by supercombinators, than is the count of the rules. We show in Table VI the count of Sequitur rules, each column represents different arity. The last column represent the actual arity of the first rule, it does not represent the amount of rules. The Book of Genesis contains more, and larger, repetitive patterns than War and Peace as we can see in Table VI.

Note, that in Table VI we have 9-ary rule for *Gen3* and *Gen6*, but no such a rule is present in *Gen12*. This is not an error, but a result of the text amount *Gen12* contains. The rule in question here is: `have dominion over the fish` **29** `sea` **79 96**. It generates the phrase `have dominion over the fish of the sea and over the fowl of the air`. In case of *Gen12*, the rule for this phrase is different: `have dominion` **531** `fish` **574 86 111**. More sub-phrase rules have been created, since they are reused elsewhere in *Gen12* text. Two 9-ary rules in *Gen* sample are unrelated, since they generate different phrases.

A. Difference between Binary and N-ary Trees.

The first experiment is going to show us, whether we should use binary[2] or n-ary trees. As we mentioned in Section IV-C, n-ary trees might result into a smaller set of supercombinators. On the other hand, a binary tree always results in supercombinators with the underlying operation having arity of maximum two. Without the merge operation in mind, the decomposition of an n-ary operation with the use of an n-ary trees would yield one supercombinator with its underlying operation having arity of n. But with the binary trees, we would get maximum of $n-1$ supercombinators with at most binary operations. The actual arity of those $n-1$ supercombinators would gradually drop from maximum of n to 2 in case that every terminal symbol is different for that particular operation. We expect larger set of supercombinators obtained from binary trees than from n-ary based on those mentioned facts.

The difference between those two approaches is illustrated in Table VII for unary and Table VIII for n-ary tree approach. The fields represent the total number of supercombinators described by their operation arity shown in the second row. As mentioned earlier, binary trees yield always supercombinators with at most binary operations. Since only the concatenation operation was used, no supercombinator with unary operation was created in both cases. Supercombinator with nullary operation is the $L^0 = \lambda x.x$. We see, that in all cases it's present only once in each set, so our merge works.

The total amount of supercombinators is dramatically different for binary and n-ary strategies. That's a direct result of the fact mentioned at the beginning of this section. Although

[2]Those trees aren't actually binary, since unary nodes are possible via *UnOp* constructor. They are called binary for brevity.

TABLE VI
THE AMOUNT OF SEQUITUR GRAMMAR RULES DIVIDED BY THEIR ARITY.

Sample	2-ary	3-ary	4-ary	5-ary	6-ary	7-ary	8-ary	9-ary	11-ary	1st rule
Gen3	122	18	2	5	-	-	-	1	-	762
Gen6	339	35	7	5	-	-	-	1	-	2116
Gen12	660	61	13	5	3	1	-	-	-	3906
Gen	3363	206	56	22	6	1	2	2	1	19512
WP1	120	6	2	-	-	-	-	-	-	1678
WP2	68	6	1	1	-	-	-	-	-	1173
WP3	73	7	-	-	-	-	-	-	-	1244
WP4	84	7	-	-	-	-	-	-	-	1189
sum	345	26	3	1	-	-	-	-	-	-
WP1-4	421	23	6	1	-	-	-	-	-	4812
WP12	1369	50	6	-	-	-	-	-	-	12582
WP24	2963	106	14	2	-	-	-	-	-	24901

TABLE VII
SUPERCOMBINATOR SETS OBTAINED FROM TWO BINARY TREE APPROACH.

Sample	0-ary	2-ary	Total
WP1	1	1682	1683
WP2	1	1173	1174
WP3	1	1246	1247
WP4	1	1191	1192
Sum	4	5292	5296
Merged	1	5272	5273
WP1-4	1	4825	4826

TABLE VIII
SUPERCOMBINATOR SETS OBTAINED FROM n-ARY TREE APPROACH.

Sample	0-ary	2-ary	3-ary	4-ary	5-ary	1st rule	Total
WP1	1	4	3	1	-	1 (1678)	10
WP2	1	3	2	1	1	1 (1173)	9
WP3	1	4	2	-	-	1 (1244)	8
WP4	1	3	2	-	-	1 (1189)	7
Sum	4	14	9	2	1	4	34
Merged	1	5	5	1	1	4	17
WP1-4	1	8	6	2	1	1 (4812)	19

all supercombinators in the binary section have at most binary operations, the real arity of them can vary. In case of the *WP1* sample, the maximum arity was 728. That is a smaller number than the actual arity of the first rule of Sequitur grammar, which is 1678. This is because the supercombinator form is non-redundant, i.e. the words (the actual arguments of supercombinators from which the total arity is calculated) do not repeat themselves.

If we compare the sum of supercombinators obtained from four different War and Peace chapters and the number of supercombinators obtained after the merge (on supercombinator level) we see no significant difference in the case of Binary trees (5296 compared to 5273). In the n-ary tree case however, the difference is an exact half (34 to 17).

In comparison with the processed grammar of four chapters together, the sample *WP1-4*, we see the difference in case of binary trees (5273 for merged against 4826 for *WP1-4* sample). In the other case however, the difference is insignificant (17 compared to 19). In case of other arities in n-ary tree case, the merged has unified mostly supercombinators with lower arity operations. The difference between merge and *WP1-4* strategy is rather insignificant.

We can thus conclude, that in the case of Sequitur generated grammars, the n-ary tree approach seems to yield significantly smaller set of supercombinators, therefore we are going to use this approach further on in this paper.

B. Constructing Supercombinator Set from a Sequitur Grammar

Straightening out the issue of what tree forms to use we can now proceed to the evaluation of larger book parts. The results are shown in Table IX.

Supercombinators are again divided by their operation arity. This is an especially useful feature for comparison with the arity of Sequitur grammars. We can see that for every arity of the original Sequitur rules (see Table VI) there exists at least one supercombinator with the same operation arity. This result was expected, since we have used n-ary trees. And the actual number of supercombinators is always lower or equal to the number of rules. We can say, that our process does not create any unexpected supercombinators that might introduce different structure into the original text. The arity of the first rule is the same as the arity of the operation within the top supercombinator, which is again the result of the n-ary tree usage.

Another interesting result is the comparison of the difference between binary rules and their respective supercombinators. Since the Book of Genesis seems to have more equal parts, as concluded in Section V, it has more supercombinators with larger arity operation. This also means that the sets of lower arities are larger than in the case of War and Peace. We see that difference in *Gen* and *WP24*, where the former has dropped from 3363 to 126 and the latter from 2963 to 33 for binary operations. This is due to the fact, that supercombinators with lower arity operations may contain sub-supercombinators with

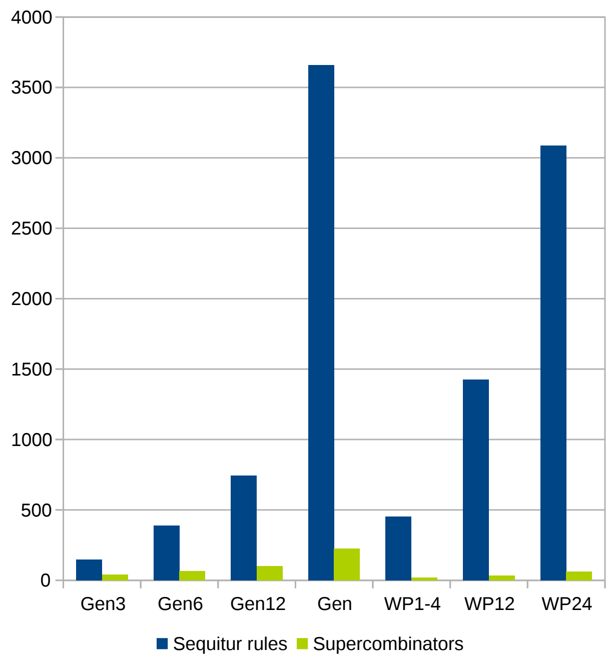

Fig. 4. Total number of Sequitur rules compared with their resulting supercombinator count.

any other operation arities. We see that *Gen* has more larger arity operations than *WP24*.

Figure 4 represents the total difference between the number of Sequitur rules and supercombinators obtained from them. The difference is rather significant. This shows, that our process is capable of data reduction. Different grammar rules with the same structure are merged into a single supercombinator, and as we can see, this has a significant impact in the case of Sequitur generated grammars.

VI. Discussion

Presented results show that our approach can prevent the structural explosion. Although this has already been pointed out in our previous work, we have performed small experiments on a simple regular grammar examples. This paper presents larger scale experiments performed on context-free grammars that support these claims.

We may conclude from the results in Table IX that if the final supercombinator set contains supercombinators with higher operation arities, the total number of supercombinators with the lower ones rises with them as is the case of *Gen* sample. We do not observe this for the *WP24* sample in such a scale, since it has less supercombinators with higher arities.

The different tree approaches that we have defined result into different supercombinator sets. Should we use only binary like trees, the resulting set of supercombinators in the case of Sequitur generated grammars would be larger and thus the promised reduction of size would be impossible. The other, list oriented approach does not suffer from that drawback and delivers the promised results.

Another interesting discovery lies with the analysis of the text itself. Although Sequitur algorithm may be useful for analysis, for example if we want to find out most occurring phrase or longest occurring repeating phrase (like the rule 78 one in Table V), our approach takes this notion one step further. By one step further we mean the analysis of the abstract structures, that supercombinators are. One of those is the argument string length of L^0 supercombinators, that presents absolute count of all words occurring in the text. This number obviously differs from the word count in Table IV, since we do not store duplicates in our structure.

The comparison of arities signifies the difference between The Book of Genesis and War and Piece, but that discovery can be inferred from pure Sequitur results, that we show in Table VI. But should we want to find out, what structure is most used, our supercombinator form is probably better suited for that question. But this is not so visible from the current results, since we have used Sequitur, and thus produced only supercombinators with the concatenation operation. Thus further research with better grammar inference method is necessary.

VII. Related Work

Our work relates with the field of grammar inference. As already mentioned in the Introduction, not only formal inference but the induction of natural languages grammar can be incorporated with our supercombinator set construction mechanism. The induction of grammar can be achieved with the use of various different methods. Onnis, Waterfall and Edelman use cognitive graphs in their model ADIOS to infer CFG in [5]. Adriaans and Van Zaanen created the model EMILE [11], where they use probabilistic methods. Klein and Manning developed model based on constituency [12] that is also capable to induce CFG from text. As our background is in the computer languages field, Stevenson and Cordy presented concise review of the state of the art in [13], where they present various methods of grammar inference.

Our supercombinator form might be practically used in tandem with ontology extraction methods as our supercombinator form of a grammar might be used to identify concepts of a certain kind. In the work of Carvalho, Almeida, Pereira and Henriques [14] we see the use of ontologies that help the concept identification. Other uses of ontologies might be for information retrieval [15], detection of concept similarity across different information media [16] or even for the detection of mental illness from written text [17].

Related methods for concept extractions include rule based approach, as Szwed used in [18]. There we can see extraction of concepts from written text. The rules used are based on Petri nets. Other related method for text analysis is summarization. Example of this method is presented by Jassem and Pawluczuk in [19]. Those methods focus on semantic side of a text, where our supercombinator approach focuses primarily on the structure. The actual meaning is treated separately, therefore we can concentrate more clearly on those separate aspects.

TABLE IX
RESULT COUNT OF SUPERCOMBINATORS DIVIDED BY THEIR OPERATION ARITY.

	0	2	3	4	5	6	7	8	9	11	Last	Total
Gen3	1	25	7	2	4	-	-	-	1	-	1 (762)	41
Gen6	1	40	14	5	5	-	-	-	1	-	1 (2116)	67
Gen12	1	53	25	12	5	3	1	-	-	-	1 (3906)	101
Gen	1	107	71	43	20	6	1	2	2	1	1 (19512)	255
WP1-4	1	8	6	2	1	-	-	-	-	-	1 (4812)	19
WP12	1	15	15	3	-	-	-	-	-	-	1 (12582)	35
WP24	1	29	20	9	2	-	-	-	-	-	1 (24901)	62

VIII. CONCLUSION

We have presented a way to represent any CFG nor-redundantly in a single set of supercombinators. The process has been described in detail, where we show it in separate steps. Many of those steps might be performed in parallel, so better computation time is achievable.

Applications of our supercombinator structure have been presented on the samples taken from literature. The Book of Genesis and War and Piece by Tolstoy were used. Since our process works only on grammars, we needed to process those samples first with Sequitur algorithm. This algorithm constructs CFG from a finite string of symbols, in our case words. We show , that our representation significantly reduces the size of entire structure, since it is non-redundant. Further on, we have discussed the possibilities of a structure analysis, that is possible to perform on our supercombinator structure.

In our future work, we would like to extract more information with the use of better inference mechanism that Sequitur algorithm brings. In the section VII we show various possible ways to induce a grammar from text samples so we would like to actually use them in our experiments to further prove the abilities of our supercombinator form construction method.

REFERENCES

[1] S. Edelman, "On the nature of minds, or: truth and consequences," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 20, no. 3, pp. 181–196, 2008. doi: 10.1080/09528130802319086. [Online]. Available: http://dx.doi.org/10.1080/09528130802319086

[2] N. Chomsky, *Syntactic Structures*. Mouton and Co., 1957.

[3] 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

[4] 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

[5] 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

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

[7] 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

[8] 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

[9] 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

[10] 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

[11] P. W. Adriaans and M. Van Zaanen, "Computational grammar induction for linguists," *Grammars*, vol. 7, pp. 57–68, 2004.

[12] D. Klein and C. D. Manning, "Corpus-based induction of syntactic structure: Models of dependency and constituency," in *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2004. doi: 10.3115/1218955.1219016 pp. 478–485. [Online]. Available: http://dx.doi.org/10.3115/1218955.1219016

[13] 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

[14] 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: 10.4230/OASIcs.SLATE.2012.239 pp. 239–253. [Online]. Available: http://dx.doi.org/10.4230/OASIcs.SLATE.2012.239

[15] J.-A. Asensio, N. Padilla, and L. Iribarne, "Information retrieval using an ontological web-trading model," in *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems*. IEEE, 2013, pp. pages 243–249.

[16] M. Wang, X. Liu, L. Huang, B. Lang, and H. Yu, "Ontology-based concept similarity integrating image semantic and visual information," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, vol. 2. IEEE, 2014. doi: 10.15439/2014F273 pp. pages 289–296. [Online]. Available: http://dx.doi.org/10.15439/2014F273

[17] P. Šaloun, "From lightweight ontology to mental illness indication," in *Scientific Conference on Informatics, 2015 IEEE 13th International*. IEEE, 2015. doi: 10.1109/Informatics.2015.7377799 pp. 9–12. [Online]. Available: http://dx.doi.org/10.1109/Informatics.2015.7377799

[18] P. Szwed, "Concepts extraction from unstructured polish texts: a rule based approach," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, vol. 5. IEEE, 2015. doi: 10.15439/2015F280 pp. 355–364. [Online]. Available: http://dx.doi.org/10.15439/2015F280

[19] K. Jassem and L. Pawluczuk, "Automatic summarization of polish news articles by sentence selection," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, vol. 5. IEEE, 2015. doi: 10.15439/2015F186 pp. 337–341. [Online]. Available: http://dx.doi.org/10.15439/2015F186