On the expressiveness of Systemic Functional Grammars

Jeremy Dohmann

July 20, 2018

Contents

1 2

2 Introduction 2

3 Context-free languages are closed under substitution 4

4 The language of a traversal 5

5 Proof by substitution 8

6 A construction 9

1

Abstract

This paper will briefly introduce Systemic Functional Grammars (SFGs), a linguistic formalism developed by Michael Halliday and commonly used in ESL teaching and early AI text planning systems, show that they are context-free and derive a construction showing how to convert an SFG to an equivalent CFG.

2 Introduction

A Systemic Functional Grammar (SFG) is a set of networks (trees) each of which is constrained by uniform structural rules, and each of which may introduce constraints of its own via a set of possible rules ("realization rules") that manipulate lexical items through the instantiation and constraining of roles (similar to variables in a programming language).

The structure of a grammar is as follows:

Basic Type Definitions

$$
\begin{aligned}
Grammar &::= (Root,\ Set(Root)) \\
Root &::= BasicClass \\
Class &::= BasicClass \mid Gate \\
BasicClass &::= (Set(Rule),\ Set(System),\ Set(Gate)) \\
System &::= Set(BasicClass) \\
Rule &::= ProdRule \mid OrderingRule \mid GroupingRule \\
ProdRule &::= (Role,\ \textbf{String}) \mid (Role,\ Class) \\
OrderingRule &::= \text{adjacency } (Role,\ Role) \mid \text{partition } (Role,\ Role) \\
GroupingRule &::= \text{conflation } (Role,\ Role) \mid \text{expansion } (Role,\ Role) \\
Gate &::= (Type,\ Set(Rule),\ Set(Class)) \\
Type &::= \text{and} \mid \text{or}
\end{aligned}
$$

A grammar consists of a grammatical stratum followed by any number of side networks and is true (in an accepting state for a given input) iff the grammatical stratum is true.

A BasicClass consists of a set of rules which must be true in order for the class to be true, a set of child systems all of which must be true in order for the class to be true, and a set of gates that it is an entry condition for all of which must be true in order for the class to be true. Thus in terms of constraints, a BasicClass item is true iff every Rule, System, and Gate defined in its sets is true.

A System contains a set of BasicClass items. The System is true iff exactly one of its children is true.

A gate is true iff either all of its entry conditions is true (an "and" gate) or if at least one of its entry conditions is true (an "or") gate.

The above define all of the structural constraints on an SFG. Taken together, the constraints can be interpreted to mean that in order for a network to be true (in an accepting state for a given input) there must be a series of paths (a traversal) from the leaves to the root, s.t. all the systems have exactly one path crossing through them, every class has a path crossing through each of its systems, and all the realization rules of every class lying on a path in the traversal (set of paths) are true.

The realization rules can be classified into three types: production rules, ordering rules, and grouping rules.

Every rule in some way manipulates roles. Roles are variables whose instantiations are scoped to a given traversal. Thus a rule Noun may be defined for the NP network, but the scope of a given instantiation of it is a single traversal of the network. For example if we had the input sequence "the dog likes the man", Noun would be instantiated once to "dog" in one traversal of NP and once to "man" in another.

The production rules instantiate roles to tokens, ordering rules specify relative orderings of rules, and the grouping roles allow roles to subsume or alias other roles.

A production rule either permits a role to be instantiated to a literal string value (this is called a "lexification") or to the result of matching another network to the input (a "preselection"). A role can only be instantiated once within the scope, thus two rules instantiating a role to two separate things can't simultaneously be true within the same traversal. An interesting thing to note about preselections is that one can preselect a class that isn't a root class, and one can preselect multiple classes at once. All this means is that there exists a traversal of the network containing those classes that sets all of those classes to true, AND we instantiate the preselecting role to the tokens consumed by that network traversal. A consequence of that is that you can't

successfully simultaneously preselect classes from different networks with the same role, and you can't successfully preselect classes from the same network which can never occur simultaneously in the same traversal (e.g. classes which are mutually exclusive). Finally, a consequence of preselections is that they permit arbitrarily deep recursion and thus permit networks to be traversed many times during a single traversal to the grammatical stratum root.

3 Context-free languages are closed under substitution

We will use the important property of CFLs that they are closed under substitution in order to prove that SFGs are context free as well. I will now introduce the notation and concepts used in Hopcroft, Motwani, and Ullman's proof of this closure property of CFLs.

A context free language can be described by a 4-tuple, a grammar, (V, T, P, S). V is the set of all symbols which appear in the grammar, either terminal or nonterminals. T is the set of terminals. P is a set of rules of the form $X \to \beta$ where $X \in V\ T$ and β is a list of symbols in V.

Note that the description above is sufficient to describe any CFL, but not all rewrite system grammars are context-free languages.

Let's say Σ is an alphabet of terminal symbols, such that all productions in language L consist only of strings of symbols in Σ. Not necessarily all symbols in Σ or combinations of symbols in Σ need be members of L.

We define a substitution s over Σ to be a set of L_i, over an alphabet Σ_i that potentially has elements of Σ - there are no restrictions over what alphabet L_i must be over - such that we can replace $a_i \in \Sigma$ with L_i. Thus for example if our alphabet is $\{0, 1\}$ and $L_0 = \{a_n b_n\}$ and $L_1 = \{aa, bb\}$. The word 01 over Σ can be substituted via the substitutions defined above to describe the language $\{a^n b^{n+2} \mid a_n b^n aa\}$.

Let's reconsider the example above except this time with the substitutions occurring over an alphabet containing the original alphabet. Let's now define s to be the following two substitutions: $L_0 = \{a_n b_n \mid 0\}$ and $L_1 = \{aa, bb \mid 1\}$. The substitution over 01 is now $\{01 \mid 0bb \mid 0aa \mid a^n b^n 1 \mid a^n b^n aa \mid a^n b^{n+2}\}$. As you can see, we are concerned with a relatively flexible class of substitution, including those which preserve subsets of the original language, or are

designed in such a way that they can choose to function as if no substitution occurs sometimes. As long as the language L_i is context-free, we don't care whether it contains a_i or any other elements of Σ for that matter.

We have now sufficiently motivated the definition of a substitution to precisely state the substitution theorem.

Substitution theorem If L is a context-free language over alphabet Σ and s is a substitution over on Σ such that $s(a)$ is a CFL for each a in Σ, then $s(L)$ is a CFL.

The proof for this theorem can be found in Introduction to Automaton Theory, Languages, and Computation by Hopcroft, Motwani, and Ullman as **Theorem 7.23**.

Rather than provide the full proof I will attempt to provide some motivation. One can visualize how it works by appealing to the structural interpretation of a CFL. Any CFL over a CFG constructs a parse tree over every word in the language. Thus all parses in L produce trees as do all parses in $L_i(a_i)$ in s. For any given parse tree in L, at any given terminal a_i in the tree, any time we substitute another tree, corresponding to performing the substitution over L_i, the result is still a tree.

This is far from a formal proof, but it helps to motivate the notion that a substitution is akin to taking any complete parse tree T in L and creating the derivative tree T' which consists of performing any substitution of a parse tree in L_i and defining T' to be a valid tree for the language $s(L)$.

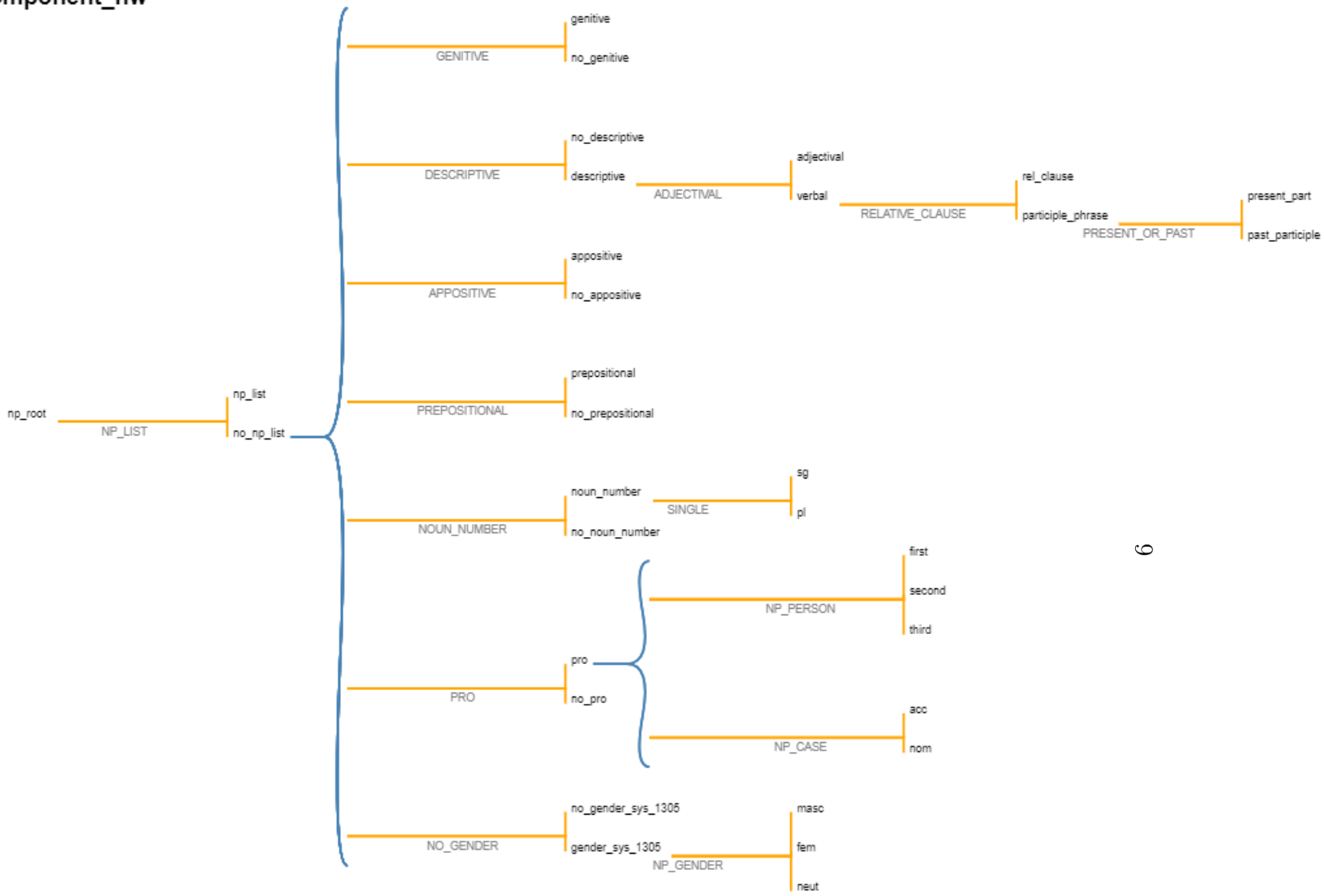
4 The language of a traversal

We now draw our attention to the structure of a network and what it means for a component network to match a string.

Structurally speaking, a network's solutions are defined by the valid traversals through it.

For example, in the grammar on the next page there three ways to traverse the gender system, 7 total ways to traverse, 3, 2, 2, 5, and 2 ways to traverse the subsequent daughter systems of no_np_list, multiplying to 840 total ways to traverse the systems of no_np_list giving 841 total paths from leaves to root satisfying all structural constraints.

np component_nw



In a given traversal, there is a unique set of role instantiations - that is, each role is instantiated only once for a valid traversal. The roles themselves can be ordered arbitrarily according to whether there are partition/adjacency constraints. So long as there are a finite number of rules, combined with the finite number of traversals, there's only a finite number of unique role instantiations and orderings for a given network.

But SFGs can recognize an infinite string set, so how can this be?

A role can be instantiated to a preselection. In fact, it can be instantiated to a simultaneous preselection of a number of classes at the same time. This looks like something of the form <np_root>for a preselection of a network and <np_root >sg,third,pronoun,acc</np_root>for a preselection with subclasses.

For now we will consider tokens of the form above to be terminals in the alphabet of a given network. Thus each network defines a language of a finite string set, limited by the number of valid traversals and the multitude of ways to order role instantiations for each traversal, where a role instantiation is always a terminal string, either corresponding to a liter terminal or a terminal string encoding some information about network/subclasses for a preselection.

It is always important to keep in mind that its possible a traversal has no valid strings recognized by it, potentially because of conflicting realization rules being applied to the same roles. E.g. if one role is instantiated to different lexical items or conflicting preselections, or if one rule puts a role in front of another and another rule does the reverse.

Therefore, the $L(\mathbb{N})$ for a network is defined to be the finite set of all arrangements of terminals described above.

We would now like to define the language of a subset of classes $\mathbb{S}$ within a network. Let's take for example the set of classes given above: sg, third, pronoun, acc. All of which are classes within the np component network in the graphic above.

The language defined by the subset $\{sg, third, pronoun, acc\}$ is analogous to the definition given for the language of the network as a whole, except that it consists only of those traversals which pass through those 4 classes (in addition any other non-conflicting classes a traversal may pass through).

Thus, just as $L(\mathbb{N})$ is a finite set of strings produced by the valid traversals of the np network, so too is $L(\mathbb{S})$ a potentially even smaller finite set of strings.

Thus the language of a set of classes is a potentially empty subset of the language of the network which contains those classes.

Notice we didn't explicitly discuss gates. Gates don't really do much besides change the topology; each network and subclass set still has a finite number of traversals and finite string set it produces.

5 Proof by substitution

If you've gotten this far, you should likely be able to tell where this proof is going. We are going to show closure under substitution allows us to start with the finite string language defined by a network and substitute the languages corresponding to the preselection terminals until we have captured the full expressiveness of the SFG.

Let's begin with some set of networks, each of which has a finite string set of productions consisting of true terminals (lexified terms) and false terminals (preselection tokens), we define our substitution to be the substitution which replaces a false terminal with the language defined by that particular network/set of subclasses, as described in the previous section.

Let us begin with our base case. Prior to performing our first substitution, all relevant networks and subclass sets described in the grammar (through preselections, etc) have a finite string language associated with them. The alphabets of any two of those languages may overlap, or may be disjoint.

Let's say we have n false terminals occurring somewhere in our grammar. We would like to perform n substitutions, at which point we will have completely described the language of our SFG, as per the specifications given in sections 2 and 4.

In the beginning we have n disjoint, finite languages, corresponding to each false terminal in our language. After performing one substitution we have replaced one symbol with a language, known to already be finite, and therefore context free. We now have $n-1$ languages and $n-1$ false terminals left to substitute. We know that the remaining $n-1$ languages are context free, because they were either inherited sans modification from the previous step, or they were the combination of substitution a known context free language into another context free language, which we showed in section 3 maintains closure.

At step i, we have performed i substitutions and there are $n-i$ false terminals left. We will perform an induction by assuming that the language described by the set of networks up until this point is all context free languages. If we perform the $i+1^{st}$ substitution, it entails unifying two languages by

taking one of our already extant languages in the set (known to be context free by hypothesis) and substituting it into another language by replacing false terminal a_{i+1} with that language. Since the two languages we are performing substitutions over are assumed to be context free, the resulting set of languages is still context free.

Thus our induction is satisfied.

6 A construction

These networks look a lot like constraint satisfaction problems, and in fact they can be represented as such. All one needs to do is represent each class, rule, and role as an integer variable according to the structural constraint, instantiation constraints, and ordering constraints defined by the rules and the topology. In order to produce the CFG defined by that SFG, all one must do is perform the construction exactly as I described. Find each traversal for each network and subclass set of interest in the grammar. Collect the string sets for each subclass set or network with some uniform encoding for false terminals, and then refer to each subclass set or network with the false terminal corresponding to it in the right hand sides of the rules.

The resulting grammar takes the exact form of the subtitution grammar I defined previously, has exactly the same expressiveness of the original SFG, and is context free.

The tremendous computational benefit is that a relatively small grammar SFG can define a context free grammar of massive proportions. Though that np component took a relatively short time to build, its CFG has well over thousands for rules. Since the rule complexity is combinatorial in the number of roles and systems, a grammar not much larger than the np described here could have many millions of rules, yet only take a relatively short amount of time to construct.

This expressiveness in producing user-defined grammars, combined with the theoretical tractability of parsing the CFG form of the grammar makes SFGs incredibly useful.

I detail elsewhere parsing algorithms which make such large grammars tractable.