Efficient generation of random sentences

Mark-Jan Nederhof

1 Introduction

Writing a grammar for a natural language is a difficult task. A number of tools have been devised to assist the author of a grammar. For example, detection of *undergeneration* (i.e. the grammar generates not all those sentences that it should) can be accomplished by automatically parsing a large corpus, and then singling out those sentences for which there is no parse according to the current version of the grammar.

For the detection of the opposite of undergeneration, viz. *overgeneration*, corpus analysis is not useful. The detection of overgeneration may be achieved however by (mechanical) random generation of sentences.[1] The grammar writer can inspect these sentences in order to find constructs which are unjustly generated by his grammar, and then make the appropriate corrections.

Random generators of sentences and sentence fragments have been incorporated into a number of development environments for grammatical formalisms [1, 2, 3, 4, 5, 6]. See [7, Chapter 8] for more playful use.

Apart from the detection of overgeneration, some other applications can be found. For example, some grammar developers use sentence generation for monitoring the consequences that alterations to a grammar have with respect to the generated language. This is done as follows. A set of sentences is generated from the grammar, once before and once after the alteration. The grammaticality of the sentences from one version of the grammar is then checked with regard to the other. This may reveal wanted or unwanted differences in the languages generated by the two versions. This is an example of *progressive evaluation* [8].

Random generation of sentences may not only be applied for testing grammars, but also for testing parsers [9, 10] and NLP systems [8]. In the area of programming languages, randomly generated programs may be used to test compilers [11, 12, 13, 14, 15].

Further use may be found in the area of natural language interfaces: a frequent complaint is that a system's linguistic capabilities are not obvious to

[1] With "random" we mean in particular that the sentences are *not* generated from some semantic value.

the user [16, Section 3]. Perhaps, automatically generated sentences may be used by the developer of the system to compose an example list of sentences within the linguistic coverage. This list is then applied to instruct users in the possibilities of the system.

This paper treats some technical aspects of the random generation of sentences and sentence fragments. We concentrate on a formalism called AGFL, which is a type of context-free grammar extended with arguments. The arguments range over finite domains, which causes useful properties related to parsing and generating to be decidable.

Our first realisations of the generator made use of naive top-down backtracking techniques, with a limit on the depth of derivations in order to ensure termination in the presence of recursion. Extensive experimentation convinced us however that the time complexities of such algorithms were unacceptably high for practical grammars, even with some techniques of smart backtracking.

Additional problems were caused by informal requirements that the generated sentences should not be too short, in order to allow interesting constructs to occur, but neither too long, in order to allow insight into the structure of the sentences.

We have solved the problem of efficiency by precomputing for each nonterminal those combinations of argument values with which derivations from that nonterminal are possible. Together with such a combination of values we also compute the minimal depth of such derivations.

The actual generation algorithm does not use any backtracking, so that it has a very low time complexity. The problem of generating sentences of convenient lengths has been solved by applying appropriate heuristics.

Because nonterminals may have very large numbers of arguments, an efficient representation for sets of tuples of argument values was required. (This representation can also be applied for *parsing*, which has been discussed in [17].)

The structure of this article is as follows. First we give an introduction to the formalism AGFL, and show that naive generation algorithms are inadequate. This is followed by a discussion of the analysis of AGFLs and the actual random generation of sentences, which uses the results of this analysis. Subsequently, an efficient representation for sets of tuples, or in other word, for subsets of Cartesian products, is discussed. We conclude by outlining possible variants and extensions of the basic analysis and generation algorithms, in two sections. The first of these sections proposes simple variants and extensions for formalisms such as AGFL. The second discusses generalization to unification-based formalisms.

2 Affix grammars over finite lattices

Affix grammars over finite lattices (AGFLs) are a restricted form of affix grammars [18, 19, 20]. An affix grammar can be seen as a context-free grammar of

which the rules are extended with *affixes* (cf. parameters, attributes, or features) to express agreement between parts of the rule. The distinguishing property of AGFLs is that the domains of the affixes are given by a restricted form of context-free grammar, the *meta grammar*, in which each right-hand side consists of one terminal.

Formally, an AGFL G is a 7-tuple $(A_n, A_t, A_p, G_n, G_t, G_p, S)$ with the following properties.

The disjoint sets A_n and A_t and the function A_p together form the *meta grammar* of G, where

- A_n is the finite set of *affix nonterminals*;
- A_t is the finite set of *affix terminals*;
- A_p is a function from elements in A_n to subsets of A_t. The fact that A_p maps some affix nonterminal A to some set of affix terminals $\{x_1, \ldots, x_m\}$ is written as

  $$A :: x_1; \ldots; x_m.$$

  We call the set $\{x_1, \ldots, x_m\}$ the *domain* of A.

The elements from A_n occur in the rules from G_p, which we define shortly, in the so called *displays*. A display is a list of elements from A_n, separated by "," and enclosed in brackets "(" and ")". Displays consisting of zero affix nonterminals are omitted.

The set of lists of zero or more elements from some set D is denoted by D^*. Thus, the set of all displays is formally described as ${A_n}^*$.

For the second part of an AGFL G we have

- G_n is the finite set of *nonterminals*. Each nonterminal has a fixed arity, which is a non-negative integer;
- G_t is the finite set of *terminals* ($G_n \cap G_t = \emptyset$);
- G_p is the finite set of *rules* $\subseteq (G_n \times {A_n}^*) \times ((G_n \times {A_n}^*) \cup G_t)^*$. Rules are written in the form

  $$N(\vec{d}) : N_1(\vec{d_1}), \ldots, N_m(\vec{d_m}).$$

  where $m \geq 0$, and N is a nonterminal, which is followed by a display $(\vec{d})$, and each member N_i, $1 \leq i \leq m$, is a terminal or nonterminal, which is followed by a display $(\vec{d_i})$. The number of affix nonterminals in a display should correspond with the arity of the symbol preceding it (we assume the arity of terminals to be zero).
- The start symbol $S \in G_n$ has arity 0.

```
PER ::  1; 2; 3.

NUM ::  sing; plur.

simple sentence:  pers pron (PER, NUM),
                  to be (PER, NUM),
                  adjective.

pers pron (1, sing):  "I".
pers pron (1, plur):  "we".
pers pron (2, NUM ):  "you".
pers pron (3, sing):  "he".
pers pron (3, plur):  "they".

to be (1,   sing):  "am".
to be (PER, NUM ):  "are", [PER = 2;
                            PER = 1 | 3, NUM = plur].
to be (3,   sing):  "is".

adjective:  "supercalifragilisticexpialadocious".
```

Figure 1: An example AGFL (with thanks to Mary Poppins)

Example 1 The small AGFL in Figure 1 illustrates some of the definitions above. This example incorporates a number of shorthand constructions, which are explained below.

The affix terminal "`sing`" in the first alternative of "`pers pron`" constitutes a so called *affix expression*, which is to be seen as an anonymous affix nonterminal, the domain of which is the singleton set {`sing`}. In general, an affix expression consists of one or more affix terminals, separated by "|", and denotes an anonymous affix nonterminal, the domain of which is the set of all specified affix terminals. E.g. the affix expression "`1 | 3`" denotes an affix nonterminal with domain {`1`, `3`}.

Another example of shorthand is the *guard* in the second alternative of "`to be`", which specifies under what conditions a form of "`to be`" is `"are"`. The comma is to be seen as "and" and the semicolon as "or". The operator "`=`" denotes unification.

The full syntax of AGFLs is defined in [21]. □ □

We use the term *variable* to refer to an instance of an affix nonterminal in an instance of a rule in a derivation (during parsing or generating).

Because the domains are finite, all possible values that variables can be instantiated with may be computed simultaneously, which reduces the amount of nondeterminism during parsing (and generating). Unification of variables now is no longer a test on equality of values, but causes an intersection of two sets of values, which should be non-empty for the unification to succeed. It should be noted that even some formal presentations of AGFLs consider variables to be set-valued.

For now, we will assume that variables are instantiated with single affix terminals. We define a *substitution* σ for a rule to be a mapping from the affix nonterminals in the rule to affix terminals from the domains of the nonterminals. If $(\vec{d})$ is some display in a rule and σ is a substitution for that rule, then $\sigma(\vec{d})$ is defined to be the tuple of affix terminals that the affix nonterminals in $(\vec{d})$ are mapped to by σ.

Later we will return to the idea that variables may be instantiated with sets of affix terminals.

3 Naive random generation

In this section, we present an algorithm to generate random sentences using a naive top-down strategy. This algorithm consists of a recursive procedure $\mathcal{G}$, which is to be called with two arguments, the first being a terminal or nonterminal and the second a tuple of affix terminals, of which the length matches the arity of the terminal or nonterminal. If it terminates, it returns a string derived from that (non)terminal with that tuple of affix terminals.

procedure $\mathcal{G}(N,t)$:
    if N *is a terminal*
    then return *the name of* N
    else **for some** *rule* $N(\vec{d}) : N_1(\vec{d_1}),\ldots,N_m(\vec{d_m})$.
                and substitution σ *for this rule*
                and tuples $t_1,\ldots,t_m$
        such that $\sigma(\vec{d}) = t$, *and*
                for $1 \leq j \leq m$ *we have* $t_j = \sigma(\vec{d_j})$
        do return $\mathcal{G}(N_1,t_1) + \mathcal{G}(N_2,t_2) + \ldots + \mathcal{G}(N_m,t_m)$
        end
    end
endproc

Figure 2: The naive generation algorithm

The procedure $\mathcal{G}$ is given in Figure 2. If the first argument is a terminal, then that terminal is the return value of the procedure. If the first argument is a nonterminal, then an appropriate rule is selected and the procedure is called recursively on the members from the right-hand side, with appropriate tuples of affix terminals. The string formed by the concatenation of the strings yielded by the recursive calls is the result of the procedure. (We denote concatenation of strings by the operator $+$.)

The nondeterminism involved in choosing a rule and an appropriate substitution can be implemented using a generator of random numbers. For example, assume that for a nonterminal N and a tuple t the grammar contains n rules of the form $N(\vec{d}) : N_1(\vec{d_1}),\ldots,N_m(\vec{d_m})$. such that a substitution σ can be found that satisfies $\sigma(\vec{d}) = t$. The procedure may then make its choice based on a random number between 1 and n. A further random number may dictate the choice of an appropriate substitution σ if more than one exists.

In order to handle cases when no rule at all can be found, a backtracking mechanism may be introduced.

Regrettably, this simple algorithm does not perform very well in practice. It may loop or require a long time to terminate. (Termination is related to consistency of probabilistic grammars [22].)

Example 2 Consider the following AGFL, which is also a context-free grammar due to the absence of affix nonterminals.

```
a:  b.
a:  "p", a.
a:  a, "q", a.

b:  a.
b:  "r".
```

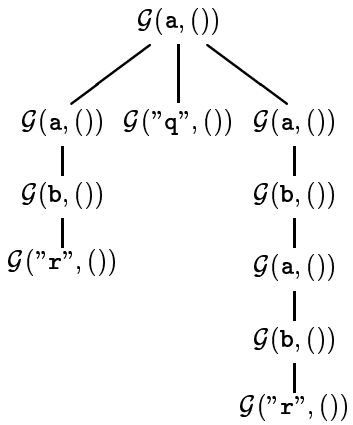

Figure 3: A call-graph for generation of `"rqr"`

An example of a string generated by a call $\mathcal{G}(\mathtt{a},())$ is given in Figure 3.

Note that if the algorithm selects one of the three rules for `a` by a probability of $1/3$ each, and similarly selects one of the two rules for `b` by a probability of $1/2$ each, then in general the chances are against termination of the algorithm.

The above grammar also demonstrates that simple refinements that introduce a bias towards selecting rules that do not lead to recursive calls cannot by themselves ensure termination, since *each* of the three rules for `a` may lead to recursion. □

Termination can be ensured by imposing a limit on the depths of derivations: an additional argument to the procedure is a counter which is increased for each level of recursion. If this counter reaches a certain threshold, the algorithm backtracks.

However, the generation process may still be very expensive, since exponentially many prospective derivations are considered within the threshold.

Example 3 Consider the following grammar.

```
a:  b, c.

b:  b, b.
b:  "p".

c:  c1.
c1:  c2.
  ...
c8:  c9.
c9:  "q".
```

For this grammar, no derivations exist of depth smaller than 11. If the threshold is set to 10, then eventually the algorithm will terminate (unsuccessfully), but many subderivations will have been computed for `b`; note that the number of subderivations for `b` is exponential in the threshold. □

4 Analysis of AGFLs

For the generation of sentences to be possible using an algorithm which is guaranteed to return a result but does not use backtracking, we need to know in advance whether a derivation is possible from some nonterminal with some affix values within some restriction on the depths of derivations. The extra restriction on the depths of derivations pertains to our previously mentioned desire to generate sentences which are not too long. It also ensures termination of the generation process.

This information for each nonterminal and each tuple of affix values can be obtained by bottom-up analysis of the grammar. The result is a set of tuples for each nonterminal, such that derivations with those tuples of affix terminals exist. The tuples are partitioned according to the minimal depth of the possible derivations.

For our algorithm, we need the following variables. For each nonterminal N we have the sets $D_i(N)$ of all tuples t such that with t a derivation of depth i has been found, but not of any depth $i' < i$; the maximum value for i that we need depends on the grammar. We also have the sets $D(N) = \bigcup_i D_i(N)$ of all tuples with which derivations of any depth have been found.

We compute the sets $D_i(N)$ for all nonterminals N, for $i = 1, 2, 3, \ldots$ in that order, based on the sets $D_{i'}(N')$, for other nonterminals N' and for $i' < i$, which have been computed earlier. We assume that initially $D_i(N) = \emptyset$ and $D(N) = \emptyset$ for all nonterminals N and indices i. For technical reasons we also have constant values $D_0(T) = \{()\}$ for all terminals T.

Figure 4 presents the algorithm in a naive way in order to enhance the simplicity of our presentation. The algorithm we actually implemented is more complicated and more efficient. In particular, we do not manipulate individual tuples but sets of tuples, as will be discussed later.

Note that we can correctly deal with rules with empty right-hand sides if we define $maximum() = 0$.

Example 4 Consider the following AGFL.

$i := 0$;
repeat $i := i + 1$;
  for each *rule* $N(\vec{d}) : N_1(\vec{d_1}), \ldots, N_m(\vec{d_m})$.
    and substitution σ *for this rule*
    and tuples $t_1, \ldots, t_m$
    and indices $i_1, \ldots, i_m$
  such that *for* $1 \leq j \leq m$ *we have*
      $t_j \in D_{i_j}(N_j)$ *and*
      $t_j = \sigma(\vec{d_j})$, *and*
    $maximum(i_1, \ldots, i_m) = i - 1$
  do $t := \sigma(\vec{d})$;
    if $t \notin D(N)$
    then $D(N) := D(N) \cup \{t\}$;
      $D_i(N) := D_i(N) \cup \{t\}$
    end
  end
until $D_i(N) = \emptyset$ *for all* N
end.

Figure 4: The analysis algorithm

```
X ::  1; 2.
Y ::  1; 2; 3.
Z ::  1; 2.
W ::  2; 3.

a(X, Z): b(X, Y), c(Y, Z).
a(X, Z): a(X, Z), d(Z).

b(X, X): "p".
c(W, Z): "q".
d(W): "r".
```

The algorithm computes

$$D_0(\texttt{"p"}) = D_0(\texttt{"q"}) = D_0(\texttt{"r"}) = \{()\}$$
$$D_1(\texttt{b}) = \{(1,1),(2,2)\}$$
$$D_1(\texttt{c}) = \{(2,1),(2,2),(3,1),(3,2)\}$$
$$D_1(\texttt{d}) = \{(2),(3)\}$$
$$D_2(\texttt{a}) = \{(2,1),(2,2)\}$$

The $D_i(N)$ for all other combinations of i and N remain $\emptyset$. □ □

procedure $\mathcal{G}(N, t, k)$:
  if N *is a terminal*
  then return N
  else for some *rule* $N(\vec{d}) : N_1(\vec{d_1}), \ldots, N_m(\vec{d_m})$.
    and substitution σ *for this rule*
    and tuples $t_1, \ldots, t_m$
    and indices $i_1, \ldots, i_m$
    such that $\sigma(\vec{d}) = t$, *and*
      for $1 \leq j \leq m$ *we have*
        $t_j = \sigma(\vec{d_j})$ *and*
        $t_j \in D_{i_j}(N_j)$ *and*
        $i_j < k$
    do *choose some numbers* $k_1, \ldots, k_m$ *such that*
      $i_j \leq k_j < k$ *for* $1 \leq j \leq m$;
      return $\mathcal{G}(N_1, t_1, k_1) + \mathcal{G}(N_2, t_2, k_2) + \ldots + \mathcal{G}(N_m, t_m, k_m)$
    end
  end
endproc

Figure 5: The efficient generation algorithm

5 Random generation of sentences

Relying on the sets $D_i(N)$ which have been computed by the algorithm from the previous section, we can generate sentences top-down without backtracking. We usually begin at the start symbol S, but we can also begin at any other nonterminal N. In the case we begin at N, we take some tuple t from $D_i(N)$, for some i. We know that derivations within some maximum depth $k \geq i$ from N with t are possible.

We determine some value $k \geq i$ using some heuristics (the higher k is chosen the deeper the computed derivation may be). We then call a procedure $\mathcal{G}$ with arguments N, t and k. This procedure is to return a string generated by some derivation from N with t, of some depth $\leq k$. (For some applications the derivation itself may be returned.)

The procedure $\mathcal{G}$ is a recursive function, which selects some rule such that derivations from the members with appropriate affix values are possible, and of which the depths are within the limit dictated by the argument k. The procedure then calls itself recursively for each member in the chosen rule with an appropriate tuple of values, and with the limit on the depths of derivations reduced by (at least) one.

Formally, $\mathcal{G}$ is defined by Figure 5.

The nondeterminism in this algorithm may be resolved using random gen-

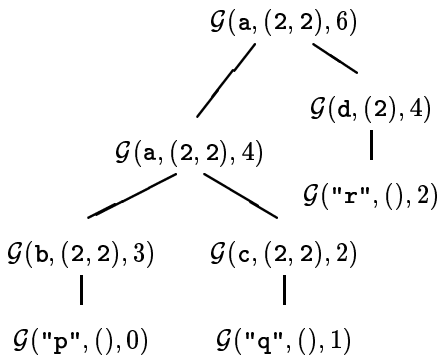

Figure 6: One possible call-graph for generation of "pqr"

erators, possibly guided by some heuristics.

Example 5 For the running example, we may call e.g. $\mathcal{G}(\mathtt{a},(2,2),6)$, since $(2,2) \in D_2(\mathtt{a})$ and $6 \geq 2$. One of the sentences which may be generated by this call is "pqr", as shown by the call-graph in Figure 6. Due to the nondeterminism of the generation algorithm, other third components in calls of $\mathcal{G}$ might have been used. For example, instead of the call $\mathcal{G}(\mathtt{d},(2),4)$, we may have had any call $\mathcal{G}(\mathtt{d},(2),k)$ for k less than 6 (the third component of the mother-node in the call-graph) but greater than or equal to 1 (the value i such that $(2) \in D_i(\mathtt{d})$).

Furthermore, other sentences, such as "pq" and "pqrr", would have been generated by $\mathcal{G}(\mathtt{a},(2,2),6)$ if other rules had been chosen in recursive incarnations of $\mathcal{G}$. □

6 Efficient processing of sets of tuples

Typically, nonterminals in AGFLs for natural languages have large arities, in some cases more than eight. Furthermore, the domains sometimes contain more than fifty affix terminals. Consequently, processing and storage of all possible combinations of affix terminals is often not feasible.

Fortunately, the set of tuples of affix terminals with which derivations exist from a certain nonterminal is typically a Cartesian product of a number of domains. (For nonterminals where this does not hold, usually a mistake in the design of the grammar can be found.) Note that a Cartesian product of domains may be represented as a list of those domains, which requires $\mathcal{O}(m \times q)$ space although $\mathcal{O}(q^m)$ tuples may be represented, where m is the arity of the nonterminal and q is the size of the largest domain.

Also for the intermediate results in our analysis and during random generation of sentences this representation may be used. In those cases however, we

sometimes need a union of a number of Cartesian products to represent a set of tuples. This amounts to the representation of a set of tuples of affix terminals by a set of tuples of sets of affix terminals.

For example, the set of tuples

$$\begin{array}{rcl} \{\ (a,b,d) & , & (a,b,e)\ , \\ (a,c,d) & , & (a,c,e)\ , \\ (p,x,y) & , & (q,x,y)\ \} \end{array}$$

can be represented by the set of tuples of sets

$$\{\ (\{a\},\{b,c\},\{d,e\}) \quad , \quad (\{p,q\},\{x\},\{y\})\ \}$$

This form can be obtained by first taking the following set of tuples of singleton sets:

$$\begin{array}{rcl} \{\ (\{a\},\{b\},\{d\}) & , & (\{a\},\{b\},\{e\})\ , \\ (\{a\},\{c\},\{d\}) & , & (\{a\},\{c\},\{e\})\ , \\ (\{p\},\{x\},\{y\}) & , & (\{q\},\{x\},\{y\})\ \} \end{array}$$

and by then merging pairs of tuples. E.g. $(\{a\},\{b\},\{d\})$ and $(\{a\},\{b\},\{e\})$ can be merged into $(\{a\},\{b\},\{d,e\})$. Repeated application of merging leads to the desired form.

The union of two sets represented in this form is complicated by the fact that two tuples may overlap. E.g. if we want to compute the union of $\{(X,Y,Z)\}$ and $\{(P,Q,R)\}$, where X,Y,Z,P,Q,R denote sets of affix terminals, we may have that $X \cap P \neq \emptyset$ and $Y \cap Q \neq \emptyset$ and $Z \cap R \neq \emptyset$. A solution is to mould e.g. $\{(P,Q,R)\}$ into a different form:[2]

$$\begin{array}{l} \{\ (P-X\ ,\ Q\ \ \ \ \ ,\ R\ \ \ \ ), \\ \ \ (P\cap X\ ,\ Q-Y\ ,\ R\ \ \ \ ), \\ \ \ (P\cap X\ ,\ Q\cap Y\ ,\ R-Z), \\ \ \ (P\cap X\ ,\ Q\cap Y\ ,\ R\cap Z)\} \end{array}$$

Possibly some of the tuples in this form may be discarded because one or more of their parts may be the empty set.

The union of the two sets is now represented by:

$$\begin{array}{l} \{\ (X\ \ \ \ \ ,\ Y\ \ \ \ \ ,\ Z\ \ \ \ ), \\ \ \ (P-X\ ,\ Q\ \ \ \ \ ,\ R\ \ \ \ ), \\ \ \ (P\cap X\ ,\ Q-Y\ ,\ R\ \ \ \ ), \\ \ \ (P\cap X\ ,\ Q\cap Y\ ,\ R-Z)\} \end{array}$$

This example demonstrates that this representation can become fragmented considerably because of the union operation (this also holds for subtraction of

[2] Note a similarity to lifting disjunction in feature descriptions [23].

sets of tuples, not shown here). To be exact, the union of k (non-overlapping) tuples of sets with m (non-overlapping) tuples of sets, respectively, may yield a set consisting of more than $k + m$ (again non-overlapping) tuples of sets, in general less than or equal to $k + m \times t^k$ tuples, where t is the width of the tuples.

To prevent quick degeneration of this representation to sets of tuples of singleton sets, it is necessary to regularly apply the merge operation, which reduces the number of tuples (possibly after deliberately fragmenting the representation further to allow more extensive merging).

We have found that the costs of merging tuples of sets do not outweight the high costs of operating on tuples of affix terminals. It required some experimenting however to determine how often the expensive merging operation had to be applied to improve instead of deteriorate the time complexity. In some cases, allowing temporary overlapping of tuples in a set proved to reduce the time-costs of the analysis.

In the present implementation, overlapping is allowed during an iteration of the analysis algorithm, i.e. for one instance of counter i. At the end of each such iteration, the sets $D_i(N)$ and $D(N)$ are restructured by eliminating the overlapping of tuples and by applying the merging operation.

A representation of sets of tuples similar to ours is proposed in [24]. This representation, called *sharing trees*, may be more compact than ours, and has the advantage that a unique sharing tree exists for each set of tuples, which simplifies the test for equivalence. However, sharing trees were developed with a certain application in mind (analysis of synchronized automata), and it seems that for our case sharing trees would not behave any better than the above representation, using sets of tuples of sets. This is because the analysis of AGFLs requires different operations (such as subtraction), some of which are more complicated for sharing trees than for our representation.

7 Simple variants and extensions

The analysis and the generation algorithm are based on the idea of computing, for each nonterminal and each tuple, the minimal depth of which derivations exist. An obvious variant results from using other quantities besides the minimal depths of derivations. We could for instance also take the minimal lengths of generated sentences from nonterminals with certain tuples. Yet other quantities, such as the number of nodes in derivations, are possible as well. For some grammars such alternative quantities may give better results.

One may further consider the use of head information [25] to influence the generation process. For example, one may choose k_j to be $k-1$ if $N_k(\vec{d_k})$ is the head of the rule $N(\vec{d}) : N_1(\vec{d_1}), \ldots, N_m(\vec{d_m})$., and i_j otherwise. This results in deep but narrow derivations, with the depth in the direction of the heads.

Such assignments to k_j may also play a role when sentences of specific forms are desired: the user may want to find sentences exhibiting a certain linguistic

phenomenon. For this, certain subderivations may need to be deep, whereas other subderivations should be short and simple since they do not directly pertain to the investigated phenomenon [8]. This idea may be realised by marking certain grammar rules that should be contained in the constructed derivations, or by allowing the user instead of the automatic generator to select the appropriate grammar rules at some critical moments [5]. After construction of a part of a derivation containing the required rules, where all k_j are chosen to be $k-1$, the remaining parts of the derivation are completed in the most simple way, by choosing all k_j to be i_j.

We have not considered the task of generating a set of sentences which is as small as possible, yet contains at least one occurrence of each rule. We refer to [9, 12, 13] for such algorithms. We feel that these algorithms, devised for the testing of compilers for programming languages, are not very useful for the detection of overgeneration of natural language grammars, since overgeneration seems to be typically caused by unforeseen interaction of several rules, instead of by individual rules. The requirement that each rule occurs at least once is therefore not sufficient. However, the general idea of increasing the coverage of a limited set of generated sentences by avoiding frequent repetition of language constructions may lead to interesting variants of our algorithm.

Other possible refinements include extension of the grammar with probabilities [11, 14, 15].

A last remark concerns an optimization of the analysis with regard to the subsequent generation. Our algorithm namely also allows generation from nonterminals other than from the start symbol S. Since from a certain nonterminal N not every other nonterminal M may be reachable, the execution of $\mathcal{G}(N,t,k)$, for some t and k may be performed without previous computation of $D(M)$ and $D_i(M)$, for many nonterminals M. In addition, the tuple t with which $\mathcal{G}$ is called may give rise to a further reduction of the search space, since not all tuples t' for a nonterminal M reachable from N may be relevant for derivations from N with t. Computing only the relevant tuples can for example be achieved by *magic transformations* [26].

8 Unification-based formalisms

The formalism of AGFL we have considered up to now has attractive properties from a computational point of view. In particular, we have seen that the minimal depth of derivations from a certain nonterminal with a certain tuple is effectively computable.

However, the grammatical formalisms that are used in practice for natural language processing are often more expressive than AGFL. An example are the unification-based formalisms, of which the definite clause grammars (DCGs) [27] are a well-known instance.

In DCGs, we have *terms* as arguments instead of affixes. A term can be

- a constant, which is comparable to an affix terminal;
- a variable, comparable to an affix nonterminal; or
- an expression of the form $f(\tau_1, \ldots, \tau_m)$, $m > 0$, where f is a *functor* and $\tau_1, \ldots, \tau_m$ are terms.

There is no equivalent to the meta grammars that we know from AGFLs, although a similar system can be derived, as shown later in this section. A nonterminal together with arguments will itself be seen as a term, by regarding the nonterminal as functor.

The language described by a grammar can be defined analogously to that in the case of AGFLs: the variables in instances of rules are instantiated with *ground* terms, which means that no variables occur within those terms. A *parse tree* formed by such a collection of rule instances derives a string in the language (cf. the tree structure in Figure 6). Two rules are linked through the left-hand side of the one and a member in the right-hand side of the other, which is allowed if they are identical after the instantiation.

For efficiency reasons however, practical implementations do not instantiate arguments further than necessary, or more precisely, variables are instantiated with terms which may contain variables themselves, until additional rule instances in the parse tree under construction provide sufficient information to further guide such instantiations effectively; cf. computations with tuples of sets of affix terminals in the case of AGFLs.

An example of a substitution is $\sigma = \{\mathtt{X}/\mathtt{g}(\mathtt{a},\mathtt{Y}), \mathtt{Z}/\mathtt{b}\}$, which means that all occurrences of `X` are to be replaced by the term $\mathtt{g}(\mathtt{a},\mathtt{Y})$ and those of `Z` are to be replaced by `b`. For example, $\sigma(\mathtt{f}(\mathtt{X},\mathtt{h}(\mathtt{Z},\mathtt{Z}))) = \mathtt{f}(\mathtt{g}(\mathtt{a},\mathtt{Y}),\mathtt{h}(\mathtt{b},\mathtt{b}))$.

We say a term τ is *more general* than τ', denoted $\tau \sqsubseteq \tau'$, if there is a substitution σ such that $\sigma(\tau) = \tau'$.

A key notion for parsing and generation with DCGs is *unification* of terms. An example is unification of two terms $\mathtt{f}(\mathtt{X},\mathtt{a})$ and $\mathtt{f}(\mathtt{g}(\mathtt{Y},\mathtt{Z}),\mathtt{Y})$, which is given by $\mathtt{f}(\mathtt{X},\mathtt{a}) \sqcup \mathtt{f}(\mathtt{g}(\mathtt{Y},\mathtt{Z}),\mathtt{Y}) = \mathtt{f}(\mathtt{g}(\mathtt{a},\mathtt{Z}),\mathtt{a})$. Informally, the term $\mathtt{f}(\mathtt{g}(\mathtt{a},\mathtt{Z}),\mathtt{a})$ is the most general term that can be obtained from both $\mathtt{f}(\mathtt{X},\mathtt{a})$ and $\mathtt{f}(\mathtt{g}(\mathtt{Y},\mathtt{Z}),\mathtt{Y})$ by a substitution. If such a unification of terms τ and τ' exists, we say τ and τ' are *unifiable*.

The substitution that instantiates the variables from two terms such that the unified term results is called the *most general unifier*. The most general unifier for terms τ and τ', if it exists, is denoted by $mgu(\tau, \tau')$. In general, $\sigma = mgu(\tau, \tau')$ implies $\sigma(\tau) = \sigma(\tau') = \tau \sqcup \tau'$. For the running example, $mgu(\mathtt{f}(\mathtt{X},\mathtt{a}), \mathtt{f}(\mathtt{g}(\mathtt{Y},\mathtt{Z}),\mathtt{Y})) = \{\mathtt{X}/\mathtt{g}(\mathtt{a},\mathtt{Z}), \mathtt{Y}/\mathtt{a}\}$.

The definition of *mgu* can be generalized to pairs of tuples of equal lengths: $mgu((\tau_1, \ldots, \tau_m), (\tau'_1, \ldots, \tau'_m))$ is the most general substitution σ such that $\sigma(\tau_1) = \sigma(\tau'_1)$, $\ldots$, $\sigma(\tau_m) = \sigma(\tau'_m)$.

The precise definition of unification is outside the scope of the present article; we refer to [28] for more explanation.

$i := 0$;
repeat $i := i + 1$;
    for each *rule* $\tau : \tau_1, \ldots, \tau_m$.
        and substitution σ
        and terms $\tau'_1, \ldots, \tau'_m$
        and indices $i_1, \ldots, i_m$
    such that *for* $1 \leq j \leq m$ *we have* $\tau'_j \in D_{i_j}$, *and*
        $\sigma = mgu((\tau_1, \ldots, \tau_m), (\tau'_1, \ldots, \tau'_m))$, *and*
        $maximum(i_1, \ldots, i_m) = i - 1$
    do $\tau' := \Phi(\sigma(\tau))$;
        if $\neg\exists\tau'' \in D[\tau'' \sqsubseteq \tau']$
        then $D := D \cup \{\tau'\}$;
            $D_i := D_i \cup \{\tau'\}$
        end
    end
until $D_i = \emptyset$
end.

Figure 7: The analysis algorithm for DCGs

8.1 Analysis of DCGs

Since the number of (ground) terms that arguments can be instantiated to is in general infinite, we need to adapt the analysis that we have performed for AGFLs. Instead of computing the minimal depths of derivations for all possible ground instantiations, we compute those depths for a finite number of tuples of arguments that have been instantiated down to a certain depth, but no further. This is achieved by the notion of *restriction* [29]. A *restrictor* is a function that truncates a term, i.e. that replaces subterms by fresh variables. A suitable restrictor for our purposes maps each term over a fixed set of functors and constants to a truncated one from a *finite* set of such terms.

A simple example of a restrictor is Φ_2, which truncates terms beyond the depth of two nested functors. For example $\Phi_2(\mathtt{f}(\mathtt{g}(\mathtt{h}(\mathtt{X},\mathtt{a}),\mathtt{i}(\mathtt{b})),\mathtt{c})) = \mathtt{f}(\mathtt{g}(\mathtt{Y},\mathtt{Z}),\mathtt{c})$. We will discuss more refined types of restrictor later.

The objective of the analysis of DCGs is to compute the minimal depths of derivations from the finite set of truncated terms. As Figure 7 shows, existing terms from D_{i_j} are combined by unifying them with members in the right-hand side of a rule. The most general unifier is then applied to the left-hand side. The result is truncated according to some suitable restrictor Φ, and matched against terms derived earlier. It is discarded if some existing term is more general; otherwise it is added to the sets D and D_i.

We initialize $D_i = \emptyset$ for all indices $i > 0$, and $D_0 = \{T \mid T \text{ is a terminal}\}$.

8.2 Generation for DCGs

The use of restrictors ensures termination of the analysis, but no longer guarantees the existence of derivations of a certain depth: if $\tau \in D_i$, for some i, then a derivation from τ of depth i may exist, but is not guaranteed to exist. The generation algorithm must therefore be able to deal with failure of the construction of a prospective derivation.

There are two obvious ways to deal with failure. First, if certain choices lead to failure, we may backtrack. This has the advantage that computation done before a "wrong" choice was made can still be used to successfully obtain a derivation. The disadvantage is that excessive (unsuccessful) computation may be needed before that choice is ever retracted. The other option is to restart the generation process from the beginning upon failure.

In our implementation we have avoided the use of backtracking as far as possible, in order to avoid the disadvantage mentioned above. Only in the process of choosing the next rule is backtracking applied. Once a rule is chosen, the algorithm is committed to that choice. When the generation process arrives in a situation where no appropriate rules at all can be found, the current attempt is abandoned and the procedure is started afresh. (In our experiments, failure occurs for about two out of three attempts to generate a sentence.)

The procedure is given in Figure 8 in an abstract form. What is not given explicitly is the backtracking needed to find some rule, a substitution, terms and indices with the required conditions. Also not shown is that all incarnations of the procedure are aborted when the above-mentioned objects cannot be found in one such incarnation.

The procedure is to be called with arguments τ' and k, where τ' is unifiable with some $\tau \in D_i$, and $k \geq i$, analogously to the generation algorithm for AGFLs that we have seen earlier.

In selecting the rule, we demand that the left-hand side τ unifies with the first parameter τ', and that the members in the right-hand side unify with terms in appropriate sets D_{i_j}; cf. Figure 5. The most general unifier σ_0 is determined, and in the first recursive call for the first member τ_1 of the rule, this substitution is applied on that member τ_1. The motivation for instantiating variables in that member at an early stage is that it may help recursive incarnations of the procedure to select only those rules that are likely to lead to successful generation. Note that there is no need to take into account the unification of the first member τ_1 itself with some τ_1' for the purpose of computing σ_0, since the variables in τ_1 will be appropriately instantiated in the first recursive call in any case.[3]

The first recursive call results in a pair (τ_1'', w_1), where w_1 is a string derived

[3]For a rule with zero members or one member in the right-hand side we have $m = 0$ or $m = 1$, respectively. In both cases the expression $mgu((\tau, \tau_2, \ldots, \tau_m), (\tau', \tau_2', \ldots, \tau_m'))$ should be read as $mgu(\tau, \tau')$. For a rule with zero members, the condition "τ_1 and $\sigma_0(\tau_1')$ are unifiable" should be read as "true".

procedure $\mathcal{G}(\tau', k)$:
  if τ' *is a terminal*
  then return (τ', τ')
  else **for some** *rule* $\tau : \tau_1, \ldots, \tau_m$.
    and substitution σ_0
    and terms $\tau'_1, \ldots, \tau'_m$
    and indices $i_1, \ldots, i_m$
    such that $\sigma_0 = mgu((\tau, \tau_2, \ldots, \tau_m), (\tau', \tau'_2, \ldots, \tau'_m))$,
      τ_1 *and* $\sigma_0(\tau'_1)$ *are unifiable, and*
      for $1 \leq j \leq m$ *we have*
        $\tau'_j \in D_{i_j}$ *and*
        $i_j < k$
    do *choose some numbers* $k_1, \ldots, k_m$ *such that*
      $i_j \leq k_j < k$ *for* $1 \leq j \leq m$;
      for $j = 1, \ldots, m$
      do $(\tau''_j, w_j) := \mathcal{G}(\sigma_{j-1}(\tau_j), k_j)$;
        $\sigma_j := mgu((\tau, \tau_1, \tau_2, \ldots, \tau_m), (\tau', \tau''_1, \ldots, \tau''_j, \tau'_{j+1}, \ldots, \tau'_m))$
      end;
      return $(\sigma_m(\tau), w_1 + \ldots + w_m)$
    end
  end
endproc

Figure 8: Generation for DCGs

from τ_1'', and $\sigma_0(\tau_1) \sqsubseteq \tau_1''$ due to possible further instantiation of variables from τ_1 in recursive incarnations. Now a substitution σ_1 is computed, this time taking into account τ_1'' instead of τ_1', and the next recursive call is made.

This is repeated until for each member in the right-hand side a string w_i has been found that is generated from that member. The substitution σ_m accumulates the results from all unifications performed recursively and is applied on the left-hand side to form $\sigma_m(\tau)$, which is returned together with the concatenated string $w_1 + \ldots + w_m$.

We have performed experiments with HPSG-style grammars [30], which were compiled to DCGs. Such grammars contain many lexical entries, which are in effect rules of the form $\tau : T.$, where T is a terminal, and τ a term. In comparison to the lexical entries, the grammar contains few other rules. This means that if lexical entries and other rules are treated on an equal footing, then the generation procedure will choose lexical entries with a very high probability, which leads to very short sentences being generated.

Our solution is the following. We choose fixed numbers a and b, where $0 < b < a$. Appropriate values can be determined empirically. At the beginning of the procedure in Figure 8, a random number r between 0 and a is generated, and if $b < r$, then lexical entries will be tried first and only then the other rules; otherwise lexical entries are tried last.

Among the rules that are not lexical entries, we further implemented a bias towards selection of rules that have not been selected very often before. Among lexical entries, a similar bias is implemented.

It is important to note that the above ideas are merely heuristics. They work well for some grammars that we have developed, but for other grammars, the analysis and generation algorithms may require changes.

8.3 Appropriate restrictors

The analysis applies a restrictor Φ to reduce the number of terms that need to be stored in the sets D_i. In order to ensure that the analysis terminates, only a finite number of such terms should be considered, and for this it is sufficient that the restrictor maps each term to a truncated term from a *finite* set of such terms. A second requirement is that the restrictor does not truncate more than necessary, since the more is truncated, the worse the precision becomes, and the larger is the chance that subsequent attempts to generate sentences will fail.

These requirements are satisfied by a restrictor that truncates subterms that are potentially unbounded in depth. What subterms possess this potential can be found by means of type inference. Not all kinds of type inference are equally precise. The algorithm we have implemented is relatively crude with respect to other published algorithms [31, 32, 33].

For each argument position we introduce a *type*, not only for arguments to nonterminals but also for arguments to functors embedded in terms in the

grammar. Such a type represents the set of terms that may occur at the argument position. We record how different types are related according to the way that the corresponding terms and subterms are related by functor and argument position.

If in a rule two subterms are represented by the same variable, the corresponding types are unified. In contrast to more advanced kinds of type inference, described e.g. in [31, 32, 33], our type unification amounts to union only; no intersection is applied. This means that unification of types T_1 and T_2 results in a type T_3 which represents at least as many terms as those represented by T_1 and T_2 together.

If types are then found to be recursive, this indicates that corresponding subterms may be unboundedly deep, and our restrictor will apply truncation of just those subterms.

Example 6 Consider the following grammar.

```
a(h(f(X)), Y): a(h(X), Y), b(g(Y)).
a(h(i), j):  "p".

b(g(j)):  "q".
```

The type system below can be inferred. Our notation stresses that the system is very similar to a meta grammar for AGFLs.

```
S ::  a(T1, T3); b(T4).
T1 ::  h(T2).
T2 ::  f(T2); i.
T3 ::  j.
T4 ::  g(T3).
```

The "super-type" `S` represents all nonterminals together with their arguments. For nonterminal `a` the arguments are of type `T1` and `T3`, respectively. A term of type `T1` is composed of a functor `h` and an argument of type `T2`. Type `T2` is recursive: it is a subtype of itself. This indicates that terms of that type may be unboundedly deep.

A restrictor Φ that truncates terms at recursive types will map a term such as `a(h(f(f(i))), j)` to `a(h(Z), j)`. □

9 Conclusions

We have presented an algorithm for AGFLs which finds random derivations without resorting to backtracking. This algorithm relies on an analysis of the grammar. The reason the algorithm always terminates is that a non-negative value is kept which decreases each time we descend the derivation under construction. The reason the algorithm cannot fail is that the results of the analysis

provide enough information to successfully guide the algorithm towards a complete derivation.

The feasibility of the analysis depends on efficient processing of sets of tuples. Experience has shown that the analysis takes less time than finding a single derivation using a top-down backtrack algorithm.

Our technique allows much flexibility: many variants are possible, and the underspecified parts of the generation algorithm can be made more precise so as to generate sentences according to certain syntactic requirements.

The technique has also been generalized to unification-based grammars. Since the analysis in this case cannot be exact without risking nontermination, the subsequent generation algorithm cannot rely on precise guidance how to find a complete derivation, and may consequently fail. However, since each attempt at constructing a derivation is cheap, it does not necessarily lead to excessive time consumption if a successful generation attempt is preceded by several unsuccessful attempts.

Acknowledgments

Cambridge University Press has kindly granted permission for reprint of a number of sections from an article with the same name, published as *Natural Language Engineering* 2(1): 1–13, 1996.

Much of the ideas in this paper were developed at the Computer Science Department of the University of Nijmegen and at the Department of Humanities Computing of the University of Groningen. The author is currently funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the VERBMOBIL Project under Grant 01 IV 701 V0.

References

[1] V.H. Yngve. Computer programs for translation. *Scientific American*, June 1962, pages 68–76.

[2] V.H. Yngve. Random generation of English sentences. In *1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, volume 1, pages 66–82, National Physical Laboratory, Teddington, Middlesex, 1962. London: Her Majesty's Stationery Office.

[3] O. Lecarme. Usability and portability of a compiler writing system. In *Methods of Algorithmic Language Implementation*, Lecture Notes in Computer Science, volume 47, pages 41–62. Springer-Verlag, 1977.

[4] L. Karttunen and M. Kay. Parsing in a free word order language. In D.R. Dowty, L. Karttunen, and A.M. Zwicky, editors, *Natural language parsing:*

Psychological, computational, and theoretical perspectives, pages 279–306. Cambridge University Press, 1985.

[5] B. Boguraev, J. Carroll, T. Briscoe, and C. Grover. Software support for practical grammar development. In *Proc. of the 12th International Conference on Computational Linguistics*, volume 1, pages 54–58, Budapest, August 1988.

[6] M.-J. Nederhof and K. Koster. A customized grammar workbench. In J. Aarts, P. de Haan, and N. Oostdijk, editors, *English Language Corpora: Design, Analysis and Exploitation*, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora, pages 163–179, Nijmegen, 1992. Rodopi.

[7] D.R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, 1979.

[8] L. Balkan, D. Arnold, and F. Fouvry. Test suites for evaluation in natural language engineering. In *Proceedings of the Second Language Engineering Convention*, pages 203–210, London, October 1995.

[9] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, 1972.

[10] C.S. Mellish. Some chart-based techniques for parsing ill-formed input. In *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 102–109, Vancouver, British Columbia, Canada, June 1989.

[11] A.J. Payne. A formalised technique for expressing compiler exercisers. *SIGPLAN Notices*, 13(1):59–69, January 1978.

[12] A. Celentano et al. Compiler testing using a sentence generator. *Software—Practice and Experience*, 10:897–918, 1980.

[13] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, July 1982.

[14] V. Murali and R.K. Shyamasundar. A sentence generator for a compiler for PT, a Pascal subset. *Software—Practice and Experience*, 13:857–869, 1983.

[15] D.L. Bird and C.U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[16] I. Androutsopoulos, G.D. Ritchie, and P. Thanisch. Natural language interfaces to databases — an introduction. *Natural Language Engineering*, 1(1):29–81, March 1995.

[17] M.-J. Nederhof and J.J. Sarbo. Efficient decoration of parse forests. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 53–78. Ellis Horwood, 1993.

[18] C.H.A. Koster. Affix grammars. In J.E.L. Peck, editor, *ALGOL68 Implementation*, pages 95–109. North Holland Publishing Company, Amsterdam, 1971.

[19] C.H.A. Koster. Affix grammars for natural languages. In *Attribute Grammars, Applications and Systems, International Summer School SAGA*, Lecture Notes in Computer Science, volume 545, pages 358–373, Prague, Czechoslovakia, June 1991. Springer-Verlag.

[20] M.-J. Nederhof. On the borderline between finite and infinite argument domains. In C.H.A. Koster and E. Oltmans, editors, *Proceedings of the first AGFL Workshop*, Technical Report CSI-R9604, Computing Science Institute, pages 17–43, University of Nijmegen, January 1996.

[21] C. Dekkers, C.H.A. Koster, M.-J. Nederhof, and A. van Zwol. Manual for the Grammar WorkBench Version 1.5. Technical Report no. 92–14, University of Nijmegen, Department of Computer Science, July 1992.

[22] T.L. Booth and R.A. Thompson. Applying probabilistic measures to abstract languages. *IEEE Transactions on Computers*, C-22(5):442–450, May 1973.

[23] A. Eisele and J. Dörre. Unification of disjunctive feature descriptions. In *26th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 286–294, Buffalo, New York, June 1988.

[24] D. Zampuniéris and B. Le Charlier. A yet more efficient algorithm to compute the synchronized product. Research Report 94/5, Institute of Computer Science - FUNDP Namur, Belgium, 1994.

[25] G. van Noord. *Reversibility in Natural Language Processing.* PhD thesis, University of Utrecht, 1993.

[26] S. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[27] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with the augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[28] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[29] S.M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 145–152, Chicago, Illinois, USA, July 1985.

[30] C. Pollard and I.A. Sag. *Information-Based Syntax and Semantics*, volume 1. Center for the Study of Language and Information, Leland Stanford Junior University, 1987.

[31] P. Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, February 1984.

[32] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10:125–153, 1991.

[33] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming, Second International Workshop*, Lecture Notes in Computer Science, volume 874, pages 281–298, Rosario, Orcas Island, WA, USA, May 1994. Springer-Verlag.