Just-in-time subgrammar extraction for HPSG

Vlado Kešelj Nick Cercone

Department of Computer Science, University of Waterloo,
Waterloo, ON N2L 3G1, Canada,
`{vkeselj,ncercone}@cs.uwaterloo.ca`,
`http://www.cs.uwaterloo.ca/~{vkeselj,ncercone}`

Abstract. We define the basic problem of subgrammar extraction for head-driven phrase structure grammars (HPSG) in the following way:

> Given a large HPSG grammar G and a set of words W, find a small subgrammar of G that accepts the same set of sentences from W^* as G, and for each of them produces the same parse trees.

The set of words W is obtained from a piece of text. Additionally, we assume that this operation is done "just-in-time," i.e., just before parsing the text. This application requires that this operation be done in an automatic and efficient way. After defining the problem in the general framework, we discuss the problem for context-free grammars (CFG), and give an efficient algorithm for it. We show that finding the smallest subgrammar for HPSGs is an NP-hard problem, and give an efficient algorithm that solves an easier, approximate version of the problem. We also discuss how the algorithm can be efficiently implemented.

Key words: HPSG, sub-grammar extraction, filtering.

1 Introduction

Recently, there has been a lot of research activity in the area of grammar modularity. Some of the motivational factors for this work are the following:

1. **managing complexity** The natural language (NL) grammars used in natural language processing are large and complex. The difficult problems are designing, creating, testing, and maintaining them. Using smaller modules that are combined into larger grammars addresses the complexity problem.
2. **parsing efficiency** Parsing with a large, wide-coverage grammar is typically not efficient. The running-time and space requirements can be reduced by quickly extracting a small subgrammar module, and then using it to parse the text.
3. **context-based disambiguation** By having a larger grammar we achieve a better coverage, but in the same time it becomes susceptible to ambiguities. Any natural language is very ambiguous, and it is well-known that humans use world-knowledge and contextual knowledge to do disambiguation. Extracting a subgrammar based on the text to be processed can be viewed as creating a context that can improve disambiguation.

This paper is mostly concerned with problem 2 (*parsing efficiency*). Further in this section, we introduce some additional problem assumptions, which distinguish this approach from the filtering techniques (e.g. [8]), which also address parsing efficiency. Using modularity to handle problem 1 (*managing complexity*) is discussed in [5], where we define grammar modules and describe how they are combined to form larger modules. Problem 3 (*context-based disambiguation*) is addressed by a different version of subgrammar extraction, in which the subgrammar does not generate the same readings for a sentence as the large grammar, but a subset of them.

Our approach is similar to the filtering techniques, which are a recognized way to improve parser's performance. However, our approach is different because we insist that the filtered, i.e., extracted, knowledge is in the form of a grammar. This approach is theoretically sound, and in practice it provides a clean interface between subgrammar-extraction part and the parser. More arguments for this separation of the subgrammar extraction and parsing will be given in the next section, where we describe the relevant work.

2 Relevant work

An important part of the HPSG subgrammar extraction is the extraction of the corresponding type sub-hierarchy out of the original hierarchy. Efficient type operations and representation of the types, which are previously discussed in Aït-Kaci et al. [1], are used in our approximate algorithm for subgrammar extraction for HPSGs.

HPSG is currently one of the most important formalisms used in computational linguistics. This unification-based formalism is successfully used in theoretical linguistics to explain many natural-language phenomena, and it is also used in practical applications. A very readable introduction into the area of unification-based grammars is given in Shieber [16]. The HPSG formalism is described in Carpenter [2], Pollard and Sag [14], and Sag and Wasow [15]. A formal definition suitable for our current experiments in question-answering is given in Kešelj [6].

The work on subgrammar extraction is currently focused on extracting a subgrammar according to a given domain corpus. Neumann [10, 12, 11, 13, 9] uses a machine learning approach to automatically extract a corpus-oriented subgrammar. If a corpus is from a specific domain, then the extracted subgrammar will be tuned towards this domain. The HPSG formalism is used, as well as the stochastic lexicalized tree grammars. A machine learning technique is not applicable in our situation, since we do a just-in-time extraction for a given piece of text. Hence, we need a more efficient method, and it cannot use a large corpus, but accepts as input a much smaller amount of text. Besides efficiency, the goal of subgrammar extraction is also reduction in the number of readings, i.e., reduction in ambiguities. In the work of Neumann, the same set of readings is obtained. We have the same approach.

Another technique relevant to our approach is filtering, which we mentioned in the previous section. This technique is described in Kiefer et al. [8]. Filtering is a technique used in parsing to speed up the search for rules and lexical entries, and so to improve the parsing speed. It is a different approach from ours because we require a clear separation between the extraction part and the parser, and a clear interface in which the extraction part extracts a well-defined grammar that is used by the parser.

One rationale for this subgrammar requirement is consistence with the modularity approach. For example, Zajac and Jan [17] present a practical approach to modularity of unification-based grammars, in which the modules are separate executable units, connected within the system in the style of the Unix pipe command. The filtering technique is not compatible with this approach, while our subgrammar-extraction method is compatible. Within the Zajac and Jan framework, we could define a subgrammar-extraction module and a parser module, and they can be connected in a serial pipe.

Subgrammar extraction and parsing are two different problems, which can be solved using different techniques. Subgrammar extraction can benefit from information retrieval and database techniques. On the other side, parsing is an inferencing process, which is amenable to theorem-proving techniques.

In our practical application to question answering [7], subgrammar extraction is done by a Perl program, while the actual parsing is done in Java. Since the output of the subgrammar extraction module is a well-formed HPSG grammar, we can easily replace the Java parser with Lisp, Prolog, or some other HPSG parser, which makes the system very flexible.

Another application where *just-in-time subgrammar extraction* plays an important role is described in [4]. It is an Internet application, where the parser is a Java applet running on the client side. A "real-world" grammar is too large to be transfered over the network. Instead, the server contains the subgrammar extraction module, which sends subgrammars to the parser at the client side.

3 Discussion

Our practical motivation for just-in-time subgrammar extraction, which is described in previous sections, leads to the following problem: Given a text segment T and a large grammar, find a smaller grammar that is guaranteed to return the same parsing results on T as the large grammar. We assume that subgrammar extraction is a fast procedure applied before parsing, so we want to avoid expensive "parsing" operations. For this reason, we ignore the word order by assuming that instead of having text segment T, only a set of words W is given. This set can be the set of words appearing in T, or it may be a superset of words appearing in T; e.g., we may include morphologic or semantic variants of the words. Additionally, it is desirable that the obtained subgrammar is the smallest possible. We start with a definition of *subgrammar relation*.

Let Σ be a finite set of words. A general grammar G is a decision procedure that given a sentence $s \in \Sigma^*$ either does not accept the sentence s, or gives a set of parses $\{(s, p_1), \ldots, (s, p_{n_s})\}$, where $p_1, \ldots, p_{n_s}$ are parse trees. We denote such set as $G(s)$. If s is not accepted by G, we define $G(s) = \emptyset$. We define the notion of *subgrammar* in the following way:

Definition 1 (Subgrammar). *Let $\mathcal{G}$ be a class of grammars over a set of words Σ. A partial order $(\mathcal{G}, \leq)$ is called a* subgrammar relation *if for any two grammars $G_1, G_2 \in \mathcal{G}$, such that $G_1 \leq G_2$, the following condition holds:*

$$(\forall s \in \Sigma^*) \quad G_1(s) \subseteq G_2(s)$$

If $G_1 \leq G_2$, where '$\leq$' is a subgrammar relation, we say that G_1 is a subgrammar *of G_2.*

An obvious consequence of the previous definition is that $G_1 \leq G_2$ implies that the language accepted by G_1 is a subset of the language accepted by G_2. Now, we can give the formulation of the *problem of subgrammar extraction*:

> Let $\mathcal{G}$ be a class of grammars over a set of words Σ, and let $\leq$ be a subgrammar relation. Given a grammar G from $\mathcal{G}$ and a finite set of words $W \subseteq \Sigma$ find a minimal grammar G_1 with respect to the relation '$\leq$' such that for any sentence $s \in W^*$ the condition $G(s) = G_1(s)$ holds.

We use the term "a minimal grammar" since there can be more than one minimal grammar. It is also possible that there are no minimal grammars satisfying the above condition.

3.1 Context-free grammars

We follow the definition of context-free grammars (CFG) as given in Hopcroft and Ullman [3]. For two CFGs $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ over the set of words Σ ($T_1, T_2 \subseteq \Sigma$), we say that G_1 is a subgrammar of G_2, and write $G_1 \leq G_2$, if the following conditions hold: $V_1 \subseteq V_2$, $T_1 \subseteq T_2$, $P_1 \subseteq P_2$, and $S_1 = S_2$. This subgrammar relation is well-defined according to definition 1 in the class of CFGs.

We can easily see that the subgrammar extraction problem for CFGs always has one unique solution. Namely, given a CFG $G = (V, T, P, S)$ and a set of words W, we have two cases:

1. No sentences from W^* are accepted by G, in which case our minimal grammar is $(\emptyset, \emptyset, \emptyset, S)$.
2. Or, there is a non-empty set of parse trees

   $$\{p : (\exists s \in W^*) \ (s, p) \in G(s)\}.$$

   Then, the minimal grammar is determined by the sets of all variables, terminals and rules that appear in those parse trees.

Of course, the "procedure" above is not a valid solution since the set W^* is infinite in general, and so can be the set of parse trees in case 2. Even if we limit the length of a sentence and assume that the set of parse trees for each sentence is finite, the above algorithm would be exponential, which is too expensive.

We present an efficient algorithm that relies on the algorithm for removing the useless symbols in Hopcroft and Ullman [3], pages 88–89. We replace the set of terminals T of the grammar G with $T \cap W$, and then apply the algorithm for removing useless variables on the grammar $(V, T \cap W, P, S)$. The resulting grammar is the minimal solution to our problem. The proof follows from the proof of the corresponding algorithm in [3]. The algorithm follows:

Algorithm: **subgrammar extraction for CFGs**

Input:	$G = (V, T, P, S)$	a large CFG
	W	a set of words
Output:	$G_1 = (V_1, T_1, P_1, S)$	the solution of the subgrammar extraction problem

1. $V_2 \leftarrow \emptyset$, $T_2 \leftarrow T \cap W$, $P_2 \leftarrow \emptyset$
2. **Repeat**
3.   *exit_flag* $\leftarrow$ true
4.   **For each** $(v \rightarrow \alpha) \in P$
5.   **If** $\alpha \in (T_2 \cup V_2)^*$ **then**
6. $P_2 \leftarrow P_2 \cup \{v \rightarrow \alpha\}$
7. $P \leftarrow P \setminus \{v \rightarrow \alpha\}$
8. $V_2 \leftarrow V_2 \cup \{v\}$
9.   *exit_flag* $\leftarrow$ false
10. **Until** *exit_flag* $=$ true
11. **If** $S \notin V_2$ **then Return** $(\emptyset, \emptyset, \emptyset, S)$
12. $V_1 \leftarrow \{S\}$, $T_1 \leftarrow \emptyset$, $P_1 \leftarrow \emptyset$
13. **Repeat**
14.   *exit_flag* $\leftarrow$ true
15.   **For each** $(v \rightarrow \alpha) \in P_2$
16.   **If** $v \in V_1$ **then**
17. $P_1 \leftarrow P_1 \cup \{v \rightarrow \alpha\}$
18. $P_2 \leftarrow P_2 \setminus \{v \rightarrow \alpha\}$
19. $V_1 \leftarrow V_1 \cup \{\text{variables in } \alpha\}$
20. $T_1 \leftarrow T_1 \cup \{\text{terminals in } \alpha\}$
21.   *exit_flag* $\leftarrow$ false
22. **Until** *exit_flag* $=$ true
23. **Return** (V_1, T_1, P_1, S)

Description. In step 1 we initialize some sets, among them the set T_2 to the intersection of T — the set of words accepted by the grammar G, and W — the set of words given for subgrammar extraction. In loop 2–10, all unproductive symbols are removed in the bottom-up direction. In step 11, we verify that the start symbol S is productive, otherwise no sentences from W^* can be accepted by the grammar and we return the empty-language grammar. In loop 13–22, we use top-down approach to remove all unreachable symbols.

Running time. It can be easily seen that the algorithm has a $O(n^3)$ running-time complexity, where n is the input size. Let us find a more precise upper bound, and discuss how the algorithm can be efficiently implemented. The set of variables V_2 and the set of terminals T_2 are represented as bit-vectors; we assume that the input sets T and W are given as alphabetically sorted lists; and the set of rules P_2 is represented as a linked list. Hence, step 1 is executed in $O(|V| + |T| + |W|)$ time.

Loop 2–10 iterates at most $|P| + 1$ times, since in each iteration, except the last one, at least one rule is moved from the set P to P_2. The set P changes during the algorithm execution (as well as other sets), so it is important to note that the numbers $|P|$, $|V|$, $|T|$, and $|W|$ denote the initial sizes of the corresponding sets. Loop 4–9 iterates at most $|P|$ times. Step 5 is executed in $O(|\alpha|)$ time, while the other lines in loop 2–10 take constant time. Hence, loop 2-10 has the running time complexity $O(|P|^2 \cdot m)$, where m is the maximal length of α, i.e., of the right hand side of all rules in the grammar. The number m is usually very small.

Similarly, loop 13–21 has the running-time complexity $O(|P|^2 \cdot m)$. Step 11 has constant time complexity, and steps 12 and 23 have $O(|V|+|T|)$ complexity. Finally, the running time complexity of the algorithm is

$$O(|V| + |T| + |W| + |P|^2 \cdot m),$$

where $m = \max\{|\alpha| : (v \rightarrow \alpha) \in P\}$.

3.2 HPSG

We follow the definition of HPSG as given in Kešelj [6]. For two HPSGs

$G_1 = (\mathsf{Atom}_1, \mathsf{Feat}_1, \mathsf{Var}, \mathsf{Type}_1, \mathsf{Init}_1, \mathsf{Rule}_1)$ and
$G_2 = (\mathsf{Atom}_2, \mathsf{Feat}_2, \mathsf{Var}, \mathsf{Type}_2, \mathsf{Init}_2, \mathsf{Rule}_2)$

over the set of words Σ — where

$$\mathsf{Type}_1 = (\mathsf{T}_1, \sqsubseteq_1), \mathsf{Type}_2 = (\mathsf{T}_2, \sqsubseteq_2),$$

and $\sqcup_1$ and $\sqcup_2$ are respective unification operations — we say that G_1 is a subgrammar of G_2, and write $G_1 \leq G_2$, if the following conditions hold:

- $\mathsf{Atom}_1 \subseteq \mathsf{Atom}_2$,
- $\mathsf{Feat}_1 \subseteq \mathsf{Feat}_2$,
- $\mathsf{T}_1 \subseteq \mathsf{T}_2$ and $(\forall t_1, t_2 \in \mathsf{T}_2)\ t_1 \sqcup_2 t_2 \in \mathsf{T}_1$ and $t_1 \sqcup_1 t_2 = t_1 \sqcup_2 t_2$
- $\mathsf{Init}_1 \subseteq \mathsf{Init}_2$, and
- $\mathsf{Rule}_1 \subseteq \mathsf{Rule}_2$.

We assume that the set of variables Var is fixed.

This subgrammar relation is well-defined according to definition 1 in the class of HPSGs.

To know whether the subgrammar extraction problem for HPSGs has a unique solution it is important to know what exactly is a parse tree. We assume that an HPSG parse tree is labeled with its initial AVM, and that each its node is labeled with the applied rule. Assuming this, we can see that the subgrammar extraction problem for HPSGs has always one unique solution. Let G be an HPSG grammar and W be a set of words. If no sentences from W^* are accepted by G, our solution is the empty HPSG:

$$(\emptyset, \emptyset, \mathsf{Var}, (\emptyset, \emptyset), \emptyset, \emptyset)$$

Otherwise, there is a non-empty set of parse trees

$$\{p : (\exists s \in W^*)\ (s, p) \in G(s)\}.$$

The minimal grammar is determined by the set of atoms, features, types, initial AVMs, and rules that appear in those parse trees. The type hierarchy has to be closed with respect to the unification operation, so the resulting type hierarchy is obtained as a set of all types appearing in the above parse trees, and by making a closure of this set with respect to the unification operation.

Similar to CFGs, the above procedure is not constructive. However, unlike CFGs it is not easy to find an efficient algorithm for the problem since the subgrammar extraction problem for HPSGs is NP-hard.

Let us prove this statement by reducing a known NP-complete problem, 3-SAT, to the problem of subgrammar extraction for HPSG. If $p_1, p_2, \ldots, p_n$ are Boolean variables, and

$$(q_{11} \vee q_{21} \vee q_{31}) \wedge \ldots \wedge (q_{1m} \vee q_{2m} \vee q_{3m}) \quad (1)$$

is a Boolean expression, where each q_{ij} ($i \in \{1,2,3\}$, $j \in \{1,\ldots,m\}$) is either p_k or $\neg p_k$ for some variable p_k ($k \in \{1,\ldots,n\}$), then the decision problem of determining whether expression (1) is satisfiable for some assignment of true or false values to the variables $p_1,\ldots,p_n$ is an instance of the 3-SAT problem. In order to show that the subgrammar extraction problem for HPSGs is NP-hard, we define an HPSG in the following way: First, for each conjunct $(q_{1j} \vee q_{2j} \vee q_{3j})$ $(1 \leq j \leq m)$ we define three HPSG rules. If q_{1j} is p_k for some $k \in \{1,\ldots,n\}$, then the first rule is:

$$\begin{bmatrix} t_j \\ \mathsf{ASGN}\text{: } \begin{bmatrix} \mathsf{P\_}k\text{: true} \end{bmatrix} \end{bmatrix} \to (q_{1j} \vee q_{2j} \vee q_{3j})$$

Otherwise, if q_{1j} is $\neg p_k$ for some $k \in \{1,\ldots,n\}$, then the first rule is:

$$\begin{bmatrix} t_j \\ \mathsf{ASGN}\text{: } \begin{bmatrix} \mathsf{P\_}k\text{: false} \end{bmatrix} \end{bmatrix} \to (q_{1j} \vee q_{2j} \vee q_{3j})$$

Similarly, if q_{2j} is $p_{k'}$ for some $k' \in \{1,\ldots,n\}$, we define the second rule to be:

$$\begin{bmatrix} t_j \\ \mathsf{ASGN}\text{: } \begin{bmatrix} \mathsf{P\_}k'\text{: true} \end{bmatrix} \end{bmatrix} \to (q_{1j} \vee q_{2j} \vee q_{3j})$$

Otherwise, if q_{2j} is $\neg p_{k'}$ for some $k' \in \{1,\ldots,n\}$, we define the second rule to be:

$$\begin{bmatrix} t_j \\ \mathsf{ASGN}\text{: } \begin{bmatrix} \mathsf{P\_}k'\text{: false} \end{bmatrix} \end{bmatrix} \to (q_{1j} \vee q_{2j} \vee q_{3j})$$

The third rule is defined in the same way.

In this way, we have defined $3m$ rules. We define one more rule:

$$\begin{bmatrix} \textit{start} \\ \mathsf{ASGN}\text{: } \boxed{1} \end{bmatrix} \to \begin{bmatrix} t_1 \\ \mathsf{ASGN}\text{: } \boxed{1} \end{bmatrix} \wedge \begin{bmatrix} t_2 \\ \mathsf{ASGN}\text{: } \boxed{1} \end{bmatrix} \wedge \ldots \wedge \begin{bmatrix} t_m \\ \mathsf{ASGN}\text{: } \boxed{1} \end{bmatrix}$$

These are the rules of an HPSG G. The set of atoms consists of all atoms appearing in the above rules, the set of features consists of all features appearing in the rules, the type lattice is a flat hierarchy consisting of the types $\perp$, *start*, t_1, t_2, $\ldots$, t_m, and the set of starting AVMs is $\{[\textit{start}]\}$.

If the expression (1) is satisfiable then it is the only string that the grammar G accepts. Otherwise, the grammar G does not accept any sentences. If we have an algorithm for the subgrammar extraction problem for HPSGs, then we define W to be the set of all symbols appearing in (1) and find the minimal subgrammar which produces the same parsing results on each expression from W^* as G. Then, if the minimal subgrammar is empty, expression (1) is unsatisfiable. Otherwise, it is satisfiable. We have proven that the subgrammar extraction problem for HPSGs is NP-hard.

Since the exact problem is too hard, we give an algorithm for an approximate solution to the problem. The algorithm is similar to the algorithm for CFGs. For the first phase of the algorithm we ignore the structure of HPSG rules, but from each rule we replace the mother AVM and each daughter AVM with the AVMs in which only root types are kept. For illustration, a rule

$$\begin{bmatrix} \textit{typeX} \\ \ldots \end{bmatrix} \rightarrow \begin{bmatrix} \textit{typeY1} \\ \ldots \end{bmatrix} \begin{bmatrix} \textit{typeY2} \\ \ldots \end{bmatrix} \ldots \begin{bmatrix} \textit{typeYn} \\ \ldots \end{bmatrix}$$

is replaced by rule:

$$[\textit{typeX}] \rightarrow [\textit{typeY1}]\ [\textit{typeY2}] \ldots [\textit{typeYn}].$$

In the same way, all AVMs in Init are replaced with simple AVMs that contain only the root type of the original AVM. For this kind of HPSG—HPSG with no features—the subgrammar extraction problem is easy. If an HPSG with no features $G = (\mathsf{Atom}, \emptyset, \mathsf{Var}, \mathsf{Type}, \mathsf{Init}, \mathsf{Rule})$ and a set of words W are given, then the following algorithm solves the subgrammar extraction problem:

Algorithm: **subgrammar extraction for HPSGs with no features**

Input: $G = (\mathsf{Atom}, \emptyset, \mathsf{Var}, \mathsf{Type}, \mathsf{Init}, \mathsf{Rule})$ an HPSG with no features
W a set of words

Output: $G_1 = (\mathsf{Atom}_1, \emptyset, \mathsf{Var}, \mathsf{Type}_1, \mathsf{Init}_1, \mathsf{Rule}_1)$ the solution of the subgrammar extraction problem

1. $\mathsf{T}_2 \leftarrow \emptyset$, $\mathsf{Atom}_2 \leftarrow \mathsf{Atom} \cap W$, $\mathsf{Rule}_2 \leftarrow \emptyset$
2. **For each** lexical rule $([t] \rightarrow \alpha) \in \mathsf{Rule}$
3.   **If** $\alpha \in \mathsf{Atom}_2^*$ **then**
4. $\mathsf{T}_2 \leftarrow \mathsf{T}_2 \cup \{t\}$
5. $\mathsf{Rule}_2 \leftarrow \mathsf{Rule}_2 \cup \{[t] \rightarrow \alpha\}$
6. **Repeat**
7.   *exit_flag* $\leftarrow$ true
8.   **For each** phrasal rule $([t] \rightarrow \alpha) \in \mathsf{Rule}$
9.   **If** $(\forall [t_1]$ in $\alpha)(\exists t_2 \in \mathsf{T}_2)\ t_1 \sqcup t_2 \neq \top$ **then**
10. $\mathsf{Rule}_2 \leftarrow \mathsf{Rule}_2 \cup \{[t] \rightarrow \alpha\}$
11. $\mathsf{Rule} \leftarrow \mathsf{Rule} \setminus \{[t] \rightarrow \alpha\}$
12. $\mathsf{T}_2 \leftarrow \mathsf{T}_2 \cup \{t\}$
13.   *exit_flag* $\leftarrow$ false
14. **Until** *exit_flag* $=$ true
15. $\mathsf{Init}_1 \leftarrow \emptyset$, $\mathsf{T}_1 \leftarrow \emptyset$
16. **For each** $[t] \in \mathsf{Init}$ **do**
17.   **If** $(\exists t_1 \in \mathsf{T}_2)\ t \sqcup t_1 \neq \top$ **then**
18. $\mathsf{Init}_1 \leftarrow \mathsf{Init}_1 \cup \{[t]\}$
19. $\mathsf{T}_1 \leftarrow \mathsf{T}_1 \cup \{t\}$
20. **If** $\mathsf{Init}_1 = \emptyset$ **then Return** $(\emptyset, \emptyset, \mathsf{Var}, (\emptyset, \emptyset), \emptyset, \emptyset)$
21. $\mathsf{Rule}_1 \leftarrow \emptyset$
22. **Repeat**
23.   *exit_flag* $\leftarrow$ true
24.   **For each** phrasal rule $([t] \rightarrow \alpha) \in \mathsf{Rule}_2$
25.   **If** $(\exists t_1 \in \mathsf{T}_1)\ t \sqcup t_1 \neq \top$ **then**
26. $\mathsf{Rule}_1 \leftarrow \mathsf{Rule}_1 \cup \{[t] \rightarrow \alpha\}$
27. $\mathsf{Rule}_2 \leftarrow \mathsf{Rule}_2 \setminus \{[t] \rightarrow \alpha\}$
28. $\mathsf{T}_1 \leftarrow \mathsf{T}_1 \cup \{t : [t]$ appears in $\alpha\}$
29.   *exit_flag* $\leftarrow$ false
30. **Until** *exit_flag* $=$ true
31. $\mathsf{Atom}_1 \leftarrow \emptyset$
32. **Repeat**
33.   *exit_flag* $\leftarrow$ true
34.   **For each** lexical rule $([t] \rightarrow \alpha) \in \mathsf{Rule}_2$
35.   **If** $(\exists t_1 \in \mathsf{T}_1)\ t \sqcup t_1 \neq \top$ **then**
36. $\mathsf{Rule}_1 \leftarrow \mathsf{Rule}_1 \cup \{[t] \rightarrow \alpha\}$
37. $\mathsf{Rule}_2 \leftarrow \mathsf{Rule}_2 \setminus \{[t] \rightarrow \alpha\}$
38. $\mathsf{Atom}_1 \leftarrow \mathsf{Atom}_1 \cup \{$words in $\alpha\}$
39.   *exit_flag* $\leftarrow$ false
40. **Until** *exit_flag* $=$ true
41. create Type_1 by restricting Type to T_1
42. **Return** $(\mathsf{Atom}_1, \emptyset, \mathsf{Var}, \mathsf{Type}_1, \mathsf{Init}_1, \mathsf{Rule}_1)$

Description. In step 1 and loop 2–5, we keep only those lexical rules that produce words from W. We also select the types used in those rules. In loop 6–13, in a bottom-up fashion similar to the CFG algorithm, we select only those phrasal rules that can be productive, and add associated types to T_2. In steps 15–20, the new set of initial AVMs Init_1 is created. If it is empty, then no sentences from W^* can be accepted, and the algorithm finishes at step 20. In loop 22–30, only reachable phrasal rules are selected using a top-down algorithm. In loop 32–40, we select reachable lexical rules.

Data structures. Before going through the algorithm and discussing its running time, let us first discuss data structures used in it. We assume that the input set of atoms and the set of words W are represented as sorted lists. The sets Atom_1 and Atom_2 are represented as bit-vectors. Conversion from one representation to another in steps 1 and 42 takes $O(|\mathsf{Atom}| + |W|)$ time.

The set of rules Rule, Rule_1, and Rule_2 are represented as linked lists.

For type representation we use simple bit-vector transitive closure encoding, as discussed in [1]. This encoding assigns a bit position to each type except for $\top$, which denotes a contradiction and it is not used in AVMs. The encoding is also used to represent sets of types T_1 and T_2. Additionally, for both sets of types we also keep a "type signature," i.e., the bit vector representing the set:

$$\{t : (\exists t_1 \in \mathsf{T}_i)\ t_1 \sqsubseteq t\},$$

where $i = 1$ or $i = 2$. This signature enables efficient evaluation of the conditions in steps 9, 17, 25, and 35. For example, the condition $(\exists t_2 \in \mathsf{T}_2)\ t_1 \sqcup t_2 \neq \top$ is evaluated in $O(|\mathsf{T}|)$ time by a bitwise AND operation between the signature of the set T_2 and the representation of the type t_1. Each step 17, 25, and 35 takes $O(|\mathsf{T}|)$ time, and step 9 takes $O(m \cdot |\mathsf{T}|)$ time, where $m = \max\{|\alpha| : [t] \to \alpha \in \mathsf{Rule}\}$.

If the input type lattice Type is not provided in this form, we can generate this type representation in $O(|\mathsf{T}|^{2.7})$ time (see [1]). We assume that the type lattice is already provided in this form.

Running-time complexity. The running time complexity of step 1 is $O(|\mathsf{Atom}| + |W| + |\mathsf{T}|)$.

Loop 2–5 iterates not more than $|\mathsf{Rule}|$ times. The condition in step 3 is verified in $O(|\alpha|)$ time. Step 4 takes $O(|T|)$ time, and steps 3 and 5 take constant time. Hence, loop 2–5 takes $O(|\mathsf{Rule}| \cdot m + |\mathsf{Rule}| \cdot |T|)$ time.

Loop 6–14 iterates not more than $|\mathsf{Rule}| + 1$ times, and loop 8–13 iterates not more than $|\mathsf{Rule}|$ times. Step 9 takes $O(m \cdot |\mathsf{T}|)$ time, and step 12 takes $O(|\mathsf{T}|)$ time. Other steps take constant time. Therefore, loop 6-14 takes $O(|\mathsf{Rule}|^2 \cdot |\mathsf{T}| \cdot m)$ time.

Set Init_1 is represented as a linked list. Hence, step 15 is executed in $O(|\mathsf{T}|)$ time.

Loop 16–19 iterates $|\mathsf{Init}|$ times. Steps 17–19 take $O(|\mathsf{T}|)$ time, so the total running time for loop 16–19 is $O(|\mathsf{Init}| \cdot |\mathsf{T}|)$, since Init and Init_1 are represented as linked lists.

Lines 20 and 21 take constant time.

Similar to loop 6–14, loop 22-30 takes $O(|\mathsf{Rule}|^2 \cdot |\mathsf{T}| \cdot m)$ time.

The complexity of step 31 is $O(|\mathsf{Atom}|)$. The complexity of loop 32–40 is $O(|\mathsf{Rule}|^2 \cdot (|\mathsf{T}| + m))$. Step 41 takes $O(|\mathsf{T}|^2)$ time.

Finally, we conclude that the running time of the whole algorithm is $O(|\mathsf{Atom}| + |W| + |\mathsf{Rule}|^2 \cdot |\mathsf{T}| \cdot m + |\mathsf{T}|^2)$, where $m = \max\{|\alpha| : [t] \to \alpha \in \mathsf{Rule}\}$.

Now, when we know how to do subgrammar extraction efficiently on this simplified HPSG, we can describe an approximate algorithm for the general HPSG subgrammar-extraction problem. Given an HPSG G and a set of words W, we first construct a simplified HPSG G_1 by removing all features, and, consequently, by simplifying all rules and initial AVMs. Then, we apply the above algorithm for HPSGs with no features on G_1 and W, and obtain an HPSG G_2. In the third step, we recover features from G back to G_2 and obtain the final HPSG G_3, which is a subgrammar of G, and produces the same parsing results on W^*. However, it is not guaranteed that the resulting grammar will be the minimal one.

Algorithm: **approximate algorithm for subgrammar extraction for HPSGs**

Input: G an HPSG

	W	a set of words
Output:	G_3	an approximate solution to the subgrammar extraction problem, such that $G_3 \leq G$ and G_3 gives the same parsing results on W^*

1. remove all features from G and obtain G_1
2. apply subgrammar extraction algorithm for HPSGs with no features to G_1 and obtain G_2
3. recover features removed in 1 to the rules in G_2 and obtain G_3

Running-time complexity. Step 1 takes $O(\text{size}(G))$ time, where $\text{size}(G)$ is the size of the grammar G, which is

$$\text{size}(G) = O(|\mathsf{Atom}| + |\mathsf{Feat}| + \\ +|\mathsf{T}|^2 + \text{size}(\mathsf{Init}) + \text{size}(\mathsf{Rule})).$$

The value $\text{size}(\mathsf{Init})$ is the count of all features, atoms and reentrancies appearing in all AVMs of the set (duplicates counted), plus the count of all types appearing in all AVMs of the set times $|\mathsf{T}|$. The size of the rule set Rule is found in the same way. According to the definition of m, $\text{size}(\mathsf{Rule}) = O(|\mathsf{Rule}| \cdot (m + |\mathsf{T}|))$ holds.

According the the analysis of the previous algorithm, step 2 takes $O(|\mathsf{Atom}| + |W| + |\mathsf{Rule}|^2 \cdot |\mathsf{T}| \cdot m + |\mathsf{T}|^2)$ time. After taking into account the definition of $\text{size}(G)$, and by separately counting lexical and phrasal rules, we see that the algorithm complexity can be also expressed as $O(\text{size}(G) \cdot |\mathsf{Rule}|)$.

Step 3 has complexity $O(\text{size}(G))$, so the total running time of the algorithm is $O(\text{size}(G) \cdot |\mathsf{Rule}|)$.

4 Conclusion and Future Work

In this paper, we have discussed the problem of *just-in-time subgrammar extraction* for HPSGs. Why the problem is important and how it is related to other relevant methods is presented. We formally define the general problem of subgrammar extraction. The problem is then defined for CFGs, and an efficient algorithm for CFGs is presented. We define the problem for HPSGs, show that it is an NP-hard problem, and give an efficient approximation algorithm for its solution.

The future work includes the evaluation of the method in the context of question answering problem, and in the context of distributed NLP [7].

References

1. H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. on Prog. Lang. and Sys.*, 11(1):115–146, Jan. 1989.
2. B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge Tracts in Th.Comp.Sci. Cambridge Univ.Press, 1992.
3. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
4. V. Kešelj. Java parser for HPSGs: Why and how. In *Proc.of PACLING'99*, pages 288–294.
5. V. Kešelj. Modular HPSG. Technical Report CS-2001-05, Dept.of CS, Univ.of Waterloo, Canada.
6. V. Kešelj. Stefy: Java parser for HPSGs, version 0.1. Technical Report CS-99-26, Dept.of CS., Univ.of Waterloo, Canada.
7. V. Kešelj. Question answering using unification-based grammar. In *Proc.of the 14th Canadian Conf. AI'2001*, 2001.
8. H. Kiefer, B.and Krieger, J. Carroll, and R. Malouf. A bag of useful techniques for efficient and robust parsing. In *Proc.of ACL-99*, pages 473–480, College Park, MD, USA, 1999.
9. G. Neumann. Automatic extraction of stochastic lexicalized tree grammars from treebanks. In *Treebanks: building and using syntactically annotated corpora*. Kluwer.
10. G. Neumann. Application of explanation-based learning for efficient processing of constraint-based grammars. In *Proc.of the 10th Conf.on AI for Applications*, San Antonio, Texas, 1994.
11. G. Neumann. Applying explanation-based learning to control and speeding-up natural language generation. In *Proc.of ACL/EACL-97*, Madrid, 1997.
12. G. Neumann. An on-line learning method to speed-up natural language processing. Technical report, DFKI GmbH, 1997.
13. G. Neumann. Learning stochastic lexicalized tree grammars from HPSG. Technical report, DFKI GmbH, 1999.
14. C.J. Pollard and I.A. Sag. *Head-Driven Phrase Structure Grammar*. Univ.of Chicago Press, 1994.
15. I.A. Sag and T. Wasow. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford, 1999.
16. S.M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes. CSLI, 1986.
17. R. Zajac and Amtrup J. Modular unification-based parsers. In *IWPT 2000, 6th Int.Wshp.on Parsing Techn.*, Trento, Italy.