Off-line compilation of Chains for Head-driven Generation with Constraint-based Grammars

Toni Tuells, German Rigau, Horacio Rodríguez

Dept. de LSI, Universitat Politècnica de Catalunya
{atuells@gcelsa.com, g.rigau@lsi.upc.es,horacio@lsi.upc.es}

Abstract. In this paper we investigate the possibility of compiling off-line the chains of lexical signs in order to improve on some known limitations of Head-Driven generation with constraint-based grammars. The method allows the detection of problematic constructions off-line and obtains substantial performance improvements over standard head-driven algorithms.

1 Introduction

Constraint-based Grammars are used for deep linguistic processing. Constraint-based Grammars that both can be used for parsing and generation are called *reversible grammars*, and their theoretical and practical aspects have been largely acknowledged; see [1],[2] ,[3]. The most widespread control strategy for generation with reversible constraint-based grammars has been the idea of *head-driven* generation, which is a control strategy almost symmetrical to *head-corner* parsing. The underlying idea of this approach is that semantic information encoded in logical forms originates mainly from lexical entries. Therefore, in order to generate from a semantic structure, heads should be predicted first using top down information. Then, the elements of the head's subcategorization list should be generated bottom-up using the rules of the grammar, until the generated semantics matches input semantics. Rules that are used in a bottom-up fashion are called *chain rules*; and we define a *chain* as a sequence of application of chain rules. The Semantic head-driven generation algorithm (SHDG) [4] and the bottom-up generation algorithm (BUG) [1] are some well known instances of algorithms following a *head-driven* control strategy.

In spite of its natural elegance, head-driven generation suffers from different drawbacks, even when generating from simple logical forms:

- Some linguistically motivated constructions and analysis may lead to *termination or efficiency problems:* empty heads and head movement, markers, raising to object constructions and words with empty semantics.

- During the generation process, variables in the semantic representation may take inappropriate values, causing over and undergeneration problems. As noted by [5], [6],[1], *some precautions have to be taken in order to guarantee that the semantics of the generated string matches the original semantics.*

Obviously, these problems must be solved to make head-driven generation suitable for practical, large scale generation systems. *Our work goes in the direction of making constraint-based grammars truly reversible tools.*

It is well known that some of the problems before mentioned (termination, efficiency, matching) are caused by uninstantiated variables during program execution. Therefore, it is an interesting idea to investigate the possibility of using off-line compilation either to adapt grammars prior to processing or to improve the efficiency of the control strategy[1]. *In this paper we investigate this possibility by compiling off-line all possible chains corresponding to the lexical signs of the grammar. This off-line compilation technique, along with a grounding analysis, improves the performance of a standard head-driven algorithm and detects problematic constructions prior to generation*[2]. We will assume a lexicalist grammar formalisms, such as HPSG [7] where lexical categories have considerable internal structure. To asses the utility of our investigation, the methods and generators described in this paper have been applied to the grammar described in [8]. This grammar follows basically HPSG and covers - admittedly, in a simplified manner- among other linguistic phenomena, coordination, control and raising verbs, passive constructions, auxiliaries, extraposition and long-distance dependencies. The original grammar uses a flat semantics encoding; to make it suitable for Head-driven generation we have adapted it to a structured semantic encoding.

The structure of rest of the paper is as follows: in section 2 we review the literature on off-line compilation of chains. In section 3 we describe the proper method that computes chains corresponding to lexical signs. In section 4 we present some applications of our method; section 5 describes our experiments with a medium-size lexicalized grammar [8]. Finally, we present our conclusions. Hereafter we will assume some familiarity of the reader with Head-Driven Generation.

1 See [12] for an excelent source of off-line compilation techniques for NLP.

2 For our purposes, grounding analysis consists in collecting information about how variables states change during program execution.

2 Related Work

The idea of off-line compilation of chains corresponding to lexical signs is not new: [9] describes a method to compile HPSG lexical entries into a set of finite-state automata. The aim of their work is at parsing efficiency: by compiling off-line all possible chains of lexical entries, many failing unifications can be avoided at run-time. [10] describe a similar method that translates HPSG into lexicalized feature-based TAG. From our perspective, [9],[10] are concerned with the computation of maximal projections of HPSG lexical entries. *In this paper, we apply the same idea to head-driven generation, though the method described here is augmented with a grounding analysis of semantic variables that helps to detect - prior to generation – problematic constructions.*

[11] describes an algorithm for efficient head-driven generation. Basically, the rules of the grammar are reorganized to reflect the predicate-argument structure rather than the surface string. The reorganization is done by compiling off-line chains corresponding to lexical entries. Once the grammar reflects the semantic structure rather than the surface string, LR parsing techniques are applied to obtain an efficient algorithm. Interestingly, his method can be seen as using the same algorithm for parsing and generation, where the grammar for generation is obtained from the grammar for parsing. However, due to the nature of LR Parsing, a grammar with a Context-free Backbone is assumed: this makes his method unsuitable for lexicalist frameworks. Obviously, one could skip the second part of his work (the application of LR parsing techniques) and apply other (parsing) algorithms: this has been done for the AMALIA system [13] , where a bottom-up parsing algorithm is applied. However, to the best of our knowledge, none of them uses a grounding analysis to predict problematic constructions.

3 Off-line Compilation of Chains

Before we describe the compilation method we make the following assumptions:

- Grammars have productions of the form :

$$X \rightarrow X_1,\ldots,\underline{X_h}\ldots X_n$$

where the X constituents include complex syntactic and semantic information and the constituent X_h is the *head* of the production.
- Only chain rules are considered.

- The method has to be applied to fully expanded lexical entries; no on-line application of lexical rules is taken into account.
- Lexical entries are of the form X → [Phon], where X includes complex syntactic and semantic information. *Phon* is the surface realization of the lexical entry.

For illustration purposes we will use the tiny grammar shown in figures 1a,1b below.

1. s(Sem)→np(SemS), vp([np(SemS],Sem)
2. s(Sem)→ s(SemO), pp([SemO],Sem).
3. vp(Scat,Sem)→ v(Scat,Sem).
4. vp([S|R],Sem)→ vp([S|[Arg|R]],Sem), Arg.
5. np(Sem)→ pn(Sem).
6. np(Sem)→ det(SemNp,Sem), nx(SemNp).
7. nx(Sem)→ n(Sem).
8. nx(Sem)→ nx(X), pp([X],Sem).
9. pp([X],Sem)→ prep([X,Y],Sem),np(Y).

Fig. 1.b. Rules of the Grammar

```
%% Lexical Entries

n(banana) → [banana].
det(X,def(X) → [the].
pn(john)    → [john].
det(X,undef(X))  → [a].
v([np(X),np(Y)],eats(X,Y)) → [eats].  % transitive reading
v([np(X)],eats(X,Y)) → [eats].            % intransitive reading
prep([X,Y],on(X,Y)) → [on].
prep([X,Y],with(X,Y)) → [with].
```

Fig. 1b Lexical Entries of the Grammar.

Note from the grammar above that the head of each production is identified by underlying the syntactic category; for example, the head daughter of rule 1 is the verbal phrase (<u>vp</u>). Note also that Prepositional phrases can be attached at sentence level (rule 2) or at noun level (rule 8). Rule 4 deals with verbal complements. A further remark about this grammar is that we find the transitive and intransitive readings of verb *eat*: we will refer to these entries extensively throughout this paper.

Intuitively, a chain of a lexical category is a sequence of rule applications which corresponds to the reflexive and transitive closure of the head relation. We now turn to the inductive definition of the chain of a lexical sign:

Definition 1 A Chain of a lexical sign X_1 is a sequence $\langle X_1 ... X_N \rangle$ such that :

- $X_1 \rightarrow$ [Phon].
- For every X_i, X_{i+1} in $\langle X_1 ... X_N \rangle$,
  $1 <= i < N$, there is a production of the form $X_{i+1} \rightarrow Y_1, ..., \underline{Y_H}, ... Y_K$ such that X_i and Y_H unify.

The crucial point is that this relation is computed bottom-up. Figure 2 below shows the computation of the chain for the intransitive reading of eat, after aplication of rules 3 and 1.:

$$\begin{tikzcd}
\text{v([np(X)],eats(X,Y))} \rightarrow \text{[eats].} \arrow[d] \\
\text{vp([np(X)],eats(X,Y))} \arrow[d] \\
\text{s(eats(X,Y))}
\end{tikzcd}$$

Fig. 2. Computing the chain for the intransitive reading of eat.

Some valid chains derived from the grammar in figure 1 are shown below. For expository purposes, we only show major syntactic categories. Furthermore, we mark with an upper index the rule from the grammar in figure 1 which has been applied to obtain that category:

1. Chain(eats) = $\langle$v, vp^3, vp^4, $sentence^1\rangle$. (transitive reading)

2. Chain(eats) = $\langle$v, vp^3, $sentence^1\rangle$. (intransitive reading)
3. Chain(banana) = $\langle$n, $nx^7\rangle$.

4. Chain(with) = {⟨prep, pp^9, nx^8⟩,⟨prep, pp^9, $sentence^2$⟩}

Note that *with* has two possible chains which correspond to the *sentence* and *nx* attachment of prepositional phrases. We provide the following simple iterative algorithm which computes all chains of a lexical entry:

```
Chains = {⟨X⟩}.

Repeat .

NewChains = { }.

For every sequence ⟨$X_1 ... X_N$⟩ in Chains, do the following:
For every production of the form $X_M \rightarrow Y_1,...,\underline{Y_H}...Y_K$ such that $Y_H$ and $X_N$ unify ,do the following:

add ⟨$X_1 ... X_N X_M$⟩ to NewChains.

add NewChains to Chains.

Until NewChains = { }.
```

Several things are noteworthy about the process just outlined:

- Chains are computed using syntactic and semantic information.
- There may be more than one chain for a given lexical entry.
- Our method computes *maximal projections* of lexical entries.

3.1 Termination Criteria

For the simple grammar in figure 1 termination of the method can be guaranteed since it is *off-line parsable.* Informally, termination can be guaranteed if for each rule application syntactic and semantic information of the mother node is identical to those of the head-daughter node minus the information used to select the non-head daughter of the rule. For example, in the HPSG head-complement schema, the list-value COMP of the mother node is the list-value COMP of the head-daughter minus the value of the non-head daughter.

In general, however, termination cannot be guaranteed. A good example is the well known head-adjunct schemata in HPSG: the syntactic information of the mother node is selected from the syntactic head-daughter, whereas the semantic information of the mother node is selected from the non-head daughter node (the adjunct). The application of our method to any adjunct would loop for the head-adjunct rule since the syntactic information of the mother node would not be sufficiently constrained. Not surprisingly, the solution to these problems is a restriction technique: a restrictor has to be defined for each rule schema. Similar problems and solutions are described in [9],[10].

3.2 Boundness Situation of Semantic Variables of a Lexical Sign

While computing the chains of lexical entries we maintain two data structures that will track how semantic variables state changes in a chain derivation. Both structure will be used to detect problematic constructions for head-driven generation.

The structure *SemVars* of a lexical sign X is a list of the variables in the semantic dimension of lexical sign X along with his boundness situation. This structure controls the coindexation of semantic variables among the head and non head daughters in a chain derivation. We represent this structure as a tuple:

$$\mathrm{SemVars}(X) = \{(V_1, \uparrow),(V_2, \downarrow),\ldots,(V_N, \uparrow)\} \qquad \mathbf{(0)}$$

The flag '↑'stands for a connected variable, whereas '↓'stands for an unconnected variable. A connected variable is a variable which gets bound after rule aplication in a chain computation with a variable of a non-head daughter.

An example will clarify this definition. Let us look at the transitive reading of *eat* in figure 1, which we repeat here for expository purposes:

$$\mathrm{v([np(X),np(Y)],eats(X,Y))} \rightarrow \mathrm{[eats].} \qquad \mathbf{(0)}$$

The initial value of SemVars(eats) is {(X, ↓),(Y, ↓)}, i.e, initially, all variables in its semantic structure are unconnected. When computing the chain for this lexical entry, we first apply rule 3, obtaining a goal of the form:

$$\text{vp([np(X),np(Y)],eats(X,Y))} \qquad \textbf{(0)}$$

Since the variables in SemVars(eats) do not get bound with any variable of a non-head daughter, their status does not change. Then, we apply rule 4, obtaining the following situation:

$$\text{vp([np(X)],eats(X,Y))} \rightarrow \underline{\text{vp}}\text{([np(X),np(Y)],eats(X,}\mathbf{Y}\text{)), np(}\mathbf{Y}\text{).} \qquad \textbf{(0)}$$

We observe that variable Y in SemVars(eats) has been bound with a variable of the non-head daughter of the rule. Therefore, SemVars(eats) is now the following:

$$\{(X, \downarrow),(Y, \uparrow)\} \qquad \textbf{(0)}$$

Now it is the turn to apply rule 1; the obtained goal is shown below:

$$\text{s(eats(X,Y))} \rightarrow \text{np(}\mathbf{X}\text{), } \underline{\text{vp(}}\text{[np(X)],eats(}\mathbf{X}\text{,Y))} \qquad \textbf{(0)}$$

Here variable X has been bound with a variable of a non-head daughter. Thus, the final situation of the Semvars(eats) structure is: {(X, ↑),(Y, ↑)}

Let us now turn to the intransitive reading of eat:

$$\text{v([np(X)],eats(X,Y))} \rightarrow \text{[eats].} \qquad \textbf{(0)}$$

Again, its initial SemVars structure is {(X, ↓),(Y, ↓)}. When computing the chain for this lexical entry, we first apply rule 3, obtaining a goal of the form:

$$\text{vp([np(X)],eats(X,Y))} \qquad \textbf{(0)}$$

Afterwards we can only apply rule 1, since the intransitive reading of eat does not have any complements:

$$s(eats(X,Y)) \rightarrow np(\mathbf{X}),\ \underline{vp(}[np(X)],eats(\mathbf{X},Y)). \qquad (\mathbf{0})$$

Here variable X has been bound with a variable of a non-head daughter; however, variable Y has not been bound during the computation of the chain. Thus, the final situation of the SemVars(eats) structure is: $\{(X, \uparrow),(Y, \downarrow)\}$.

3.3 Non instantiated variables in Non head Daughters

So far we have seen that structure SemVars indicates whether a variable in the semantic dimension of a lexical sign is going to bound a variable of a non-head daugher during the execution of a chain. Now we will concern us with a different problem, namely, whether during the execution of a chain, a non instantiated variable of a non-head daughter shows up. Consider the following infelicitious lexical entry:

$$v([np(Z),np(Y)],read(X,Y)) \rightarrow [reads]. \qquad (\mathbf{0})$$

The chain for this lexical entry would be the same as the chain for the transitive reading of eat. After applying the rule 1, we would end up with a situation like the following:

$$s(eats(X,Y)) \rightarrow np(Z),\ \underline{vp(}[np(Z)],eats(X,Y) \qquad (\mathbf{0})$$

Note that this situation indicates that the generator would try to generate a non instantiated np. *Therefore, we enrich our chains structure with information about the degree of instantiation of non-head daughters variables*. As a result, chains look now like the following:

$$\langle(X_1,-),(X_2,+)...(X_{N},+)\rangle \qquad (\mathbf{0})$$

where '-' indicates non instantiated variables in non head daughters; '+' indicates fully instantiated variables in non head daughters. Of course, one is tempted to derive all the information related to the boundness degree of variables by inspecting lexical signs only. However, caution has to be taken with this approach. It is perfectly possible to have a non bound lexical variable that may get bound after applying a chain rule. A look at a rule for simple NP formation will clarify this point. Assume the following skeletal lexical entries:

(cat: noun, sem: rel: house, sem: def: X) → [house]. **(0)**
(cat:det , sem:def:yes) → [the].

and the following chain rule:

rule NP formation **(0)**
(cat:np,sem:S)→(cat:noun,sem:S,sem:def:D),(cat:det,sem:def:D).

It is clear that by inspecting solely the lexical entry for 'house' one cannot conclude whether a variable is going to be used or not. Variable 'DEF' is not used in the lexical entry for house, but it gets bound after applying the rule on NP formation.

4 Applications

In this section we present some applications of the previously shown method described to some known problems in Head-Driven Generation.

4.1 Preventing Over and Under Generation

Overgeneration has been defined as the production of sentences whose semantics is more specific than input semantics, and undergeneration has been defined as the production of sentences whose semantics is less specific than input semantics [5]. Following these definitions, a *correct* generator produces sentences whose semantics *matches* exactly with the input semantics. Matching is defined in terms of mutual subsumption between input and output semantics. Of course, an *incorrect* generator (i.e, a generator that produces sentences whose semantics do not match exactly input semantics) is generating sentences which are simply wrong. As reported in [4],[5] *con-*

straint-based generators follow the common practice of using the metalanguage (Prolog, for example) variables for object language variables in the semantic representation, which may lead to unwanted unification of variables taking inappropriate values.

Consider the lexical entries in figure 1 for the transitive and intransitive alternation of verb *eat*, and the following input semantics for *John eats a banana* : eats(john,banana). Both entries would qualify as lexical heads since they unify with input semantics. However, only the transitive one had to. As noted by [4],[5], a simple way to prevent unwanted unifications would be to ground our semantic representations. If the lexical entry for the intransitive entry for *eat* looked like the following:

vp([np(X)],eats(X,**23**)) (**0**)

where **23** has to be understood as a fresh atom, then we would avoid the problem.

Assuming that the grounding process should be done automatically, *how can we detect the variables to be ground* ? The structure *SemVars* in section 2 provides the source of the necessary information to ground our variables; we refer to the the slogan : ‘a variable which is not going to get bound is a good candidate to get ground’. We have seen in section 2 that the *SemVars* structure for the intransitive lexical entry of eat is the following:

$$\{(X,\uparrow),(Y,\downarrow)\} \quad (\mathbf{0})$$

Therefore, we observe that variable Y should be grounded.

4.2 Avoiding Failing Unifications

Unification is the most expensive operation performed in constraint-based frameworks [14]; therefore it is an interesting issue to avoid failing unifications by applying methods cheaper than unification. A crucial step in head-driven generation is the selection of the chain rules that connect lexical entries to the original semantics. The connection is done by selecting the appropriate chain rules,i.e. those rules whose semantic and syntactic features of the head-daughter node unify with the semantics of the lexical entry. Instead of applying each chain rule in turn, a straightforward application of our method consists in applying only those rules that appear in the chain derivation of a lexical entry. The results of this experiment are shown in next section.

5 Evaluation

We have tested two versions of the BUG algorithm with the medium-size lexicalized grammar described in [8]. The first version of the algorithm was the the standard (non-deterministic) version . The second (more deterministic) version uses off-line compilation of chains. The grammar follows basically HPSG and covers a wide range of linguistic phenomena, including control and raising verbs, passive constructions, auxiliaries and long-distance dependencies. It contains about 1200 full-fledged lexical entries and 6 rule schemata. We have tested the performance of the two algorithm on 30 sentences; results are given below (average time per sentence):

Generator	**Msec / sentence**
Standard BUG	467
Deterministic BUG	278

Mean string length was 5.5 words per sentence. On the other hand, the method correctly predicted problematic constructions related to object to raising constructions and transitive/intransitive alternations (like verb to eat).

6 Conclusion

The off-line compilation technique described here treats some well known limitations on Head-driven generation on a uniform basis. It has several advantages for generating with contraint-based grammars:

1. Problems related to uninstantiated variables occurring in run time can predicted off-line. Thus, some adaptations prior to processing can be made.
2. Efficiency is improved compared to the standard BUG algorithm.
3. The method is especially suitable for lexicalist frameworks, where lexical entries have considerable internal structure. Note that in lexicalist frameworks syntactic covariation is expressed in different lexical entries rather than in multiple grammar rules. Thus, there will be one or few chains for each lexical entry.
4. The method is compatible with other techniques designed to improve efficiency (memoization, chart generation,...).

We believe that our approach is a contribution to making contraint-based grammars true reversible tools.

References

1. van Noord G. (1993) Reversibility in Natural Language Processing , PhD Thesis, University of Utrecht.
2. Strzalkowski, T. (editor) (1994). *Reversible Grammar In Natural Language Processing*. Kluwer Acaedmic Publisher, Dordrecht, The Nederlands.
3. Neumann, G. (1998).
4. Shieber S., van Noord G., Pereira F. and Moore,R. (1990) 'Semantic-Head-Driven Generation', Computational Linguistics, vol 16(1),30-43.
5. Wedekind J. (1988). Generation as structure driven derivation. In Proceedings of Coling-88, Budapest, Hungary.
6. Gerdemann D. and Hinrichs E. (1996). Some Open Problems in Head-Driven Generation. In Linguistics and Computation, CSLI.
7. Pollard ,C. and Sag,I. (1994) Head-Driven Phrase Structure Grammar, Chicago University Press, Chicago and CSLI Publications, Stanford.
8. Sag I. and Wasow T. (1999) Syntactic Theory: A Formal Introduction, CSLI Publications
9. Torisawa,K. and Tsujii, J. (1996) 'Computing Phrasal-Signs in HPSG prior to Parsing'. In Proceedings of Coling-96, Copenhagen, Denmark.
10. Kasper, R., Kiefer,B. and Netter, K. (1995). 'Compilation of HPSG to TAG',. In Proceedings of ACL-95.
11. Samuelsson, C. (1995). 'An efficient algorithm for surface generation'. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Montreal, Canada.
12. Minnen G. (1998) Off-line Compilation for Efficient Processing with Constraint-logic Grammars, PhD Thesis, University of Tübingen.
13. Shuly W., Gabrilovich E., and Francez N. (1997). AMALIA – a unified platform for parsing and generation. In Recent Advances in Natural Language Processing (RANLP-97), Tzigoz Chark, Bulgaria.
14. Kiefer B., Krieger H., Carroll J. and Malouf R. (1999). A Bag of Useful Techniques for Efficient and Robust Parsing. In Proceedings of ACl 1999.