Precompilation of HPSG in ALE into a CFG For Fast Parsing

John C. Brown
INVU Services Ltd.,
Blisworth Hill Farm,
Stoke Road,
Blisworth, UK, NN7 3DB
johnbrown@research.softnet.co.uk

Suresh Manandhar
Department of Computer Science,
University of York,
York, UK, YO1 5DD
Suresh@cs.york.ac.uk

Abstract

Context free grammars parse faster than TFS grammars, but have disadvantages. On our test TFS grammar, precompilation into CFG results in a speedup of 16 times for parsing without taking into account additional mechanisms for increasing parsing efficiency. A formal overview is given of precompilation and parsing. Modifications to ALE rules permit a closure over the rules from the lexicon, and analysis leading to a fast treatment of semantic structure. The closure algorithm, and retrieval of full semantic structure are described.

Introduction

Head Driven Phrase Structure Grammar (HPSG), Pollard and Sag (1994) is expressed in Typed Feature Structures (TFSs). Context Free Grammar (CFG) without features supports much faster parsing, but a TFS grammar has many advantages. Fast parsing can be obtained by precompiling a CFG approximation, with TFSs converted into CF near-equivalents. CFG parsing eliminates impossible trees, and TFS unification over the remainder eliminates more, and instantiates path values. Our method treats slashes separately in a precompiled table, and careful allocation of categories to TFSs makes TFS unification unnecessary: instead skeleton semantic structures are formed in parsing, and full structures retrieved afterwards.

A prototype precompiler and fast parser[1] were built in Prolog, and tested with an HPSG grammar of English by Matheson (1996), written in the ALE formalism, by Carpenter and Penn (1996). This has the 6 schemas and 5 principles of HPSG, with 184 lexemes.

A complex sentence, *"kim can believe sandy can expect sandy to persuade kim to promise sandy to try to expect sandy to persuade kim to promise kim to give sandy a happy happy book*", parsed in 3.3s. with retrieval, 5.6 times faster than with TFSs, Brown and Manandhar (2000). An 11 word sentence was 18 times faster at 87ms., or 16 times counting retrieval.

1 Relationship to Work Elsewhere

In precompilation, Torisawa et. al. (2000) repeatedly applied rules leading to maximal projections from lexical heads, allocating categories to mothers and non-head daughters. Our approach allocates categories in a closure over the rules starting with the lexicon, as in Kiefer and Krieger (2000). We differ from both these in that the CFG grammar equates to the TFS grammar, accepting exactly the same strings: a CFG parse tree translates into a TFS tree with no loss of nodes.

We precompiled 550 CF categories and 18,000 rules in 1 hour on a 280MHz. Pentium II. This compares to 5,500 categories and 2,200,000 rules from 11,000 lexemes in 45 hours by Kiefer and Krieger on a 300MHz. Sun Ultrasparc 2, where sets of TFSs in the closure are replaced by common most-general unifiers. The common unifier technique was used to add a CFG back-bone to a unification grammar by Carroll (1993). An `np` has the same *number*, *person* and *gender* features irrespective of an optional specifier and multiple adjectives. A lexical `vp`, partially saturated `vp` and sentence contain decreasing subcategorisation data. Therefore numbers of phrasal and lexical categories are comparable,

[1] Downloadable code on http://www.cs.york.ac.uk/~johnb or http://www.soft.net.uk/research/hsppar.htm .

even with exact CF representation. Since this eliminates Kiefer and Krieger's annotation with values of *relevant paths*, large rule numbers are more tolerable.

Our parsing speed-ups are comparable with Kentaro Torisawa's speed-ups of between 47 and 4 in C++ with arrays to store rules and edges, on a 300MHz. Sun Ultrasparc. We used Prolog without constant-time access arrays, with clauses hash-indexed on just the first argument. Global data structures for edges in a chart necessitate dynamic clauses with heavy assertion costs: when referenced, all arguments are treated before matching any against precompiled tables. Tree-structured tables are inefficient, using either one clause per arc with heavy invocation costs, or arguments of nested structures, copied on clause invocation. From full instrumentation revealing bottlenecks, we predict a further speed-up of 10 times, using ideal global data structures in imperative code.

2 Formal View of Precompilation and Parsing

Our CF and TFS grammars are equivalent for two reasons. First, in CF category allocation, TFSs in the list reached by NONLOCAL: INHERITED: SLASH are allocated separate categories and are otherwise ignored. The *filler-head* schema unifies a list member with a TFS from the first daughter, so a precompiled table can predict the outcome during CF parsing. This avoids over-generation of categories formed by alternative slash TFSs in multiple phrasal TFSs arising in long-range dependencies. Second, although semantic sub-structure (also containing word morphology) is omitted in category allocation, its *index* structure may be considered depending on the HEAD value. This makes CFG categories maximally selective. The now unnecessary TFS unification over the CFG parse tree is replaced by semantic TFS retrieval from skeleton structures in constituents.

The method treats grammar rules where mother and daughters each comprise a TFS, expressed as conjoined constraints :

$$TFS_0 \rightarrow TFS_1 \ldots TFS_n \tag{1}$$

where a TFS is given by:

$$\begin{aligned}
&\langle TFS \rangle \rightarrow \langle FS \rangle \mid \langle type \rangle \,\&\, \langle FS \rangle \mid \langle type \rangle \\
&\langle FS \rangle \rightarrow \langle path \rangle : \langle value \rangle \mid \langle FS \rangle \,\&\, \langle FS \rangle \\
&\langle value \rangle \rightarrow \langle TFS \rangle \mid \langle variable \rangle \,\&\, \langle TFS \rangle \\
&\langle path \rangle \rightarrow \langle feature \rangle \mid \langle feature \rangle : \langle path \rangle
\end{aligned} \tag{2}$$

The value following a *feature* in a TFS of a particular *type* is a sub-type of that given by an appropriateness function $Approp : T \times F \rightarrow T$ where F and T denote finite sets of feature symbols and types. T is organised into a subsumption hierarchy with a single root *bot*.

The grammar may either be lexicalist, or have a large number of specialised rules. Each TFS is *well-formed*, containing no feature not appropriate to its *type*: it is not *totally well-formed* with all appropriate features present. TFS unification involves *type-coercion* which also occurs in constraint application to a TFS, for example of $feature : value$ in a rule. Where a TFS of type t_1 does not contain *feature,* it is coerced into the most general sub-type t_2 of t_1 for which $Approp(t_2, feature)$ is defined.

An integer T_l, corresponding to a CFG category, is allocated in our precompilation to each unique lexical TFS_l, ignoring nested TFSs encoding slashes and semantics, so that

$$\begin{aligned}
T_l = \text{retrieve}^{-1}(&\text{omit}(slash : v1, \\
&\text{re-introduce}(index : v2, \\
&\text{omit}(semantics : v3, TFS_l))))
\end{aligned} \tag{3}$$

where *slash, semantics* and *index* are paths and $v1$, $v2$ and $v3$ are values. Implementation is by matching against a discrimination tree which is traversed in correspondance with TFS_l, from which types are ignored in a subtree reached by *path* where $\text{omit}(path : v, TFS_l)$ applies and considered where $\text{re-introduce}(path : v, TFS_l)$ applies. Where corresponding types from TFS_l and the tree are unequal, a new branch is grown, terminating with a new T_l: this mechanism ensures each terminal is marked. Because co-indexing by syntactic paths in a rule (Section 5) unifies *index*, this is re-introduced in an `np` and in a category that unifies with an `np` to form an `np`. The function in (3) is an example

of a *restrictor*, Shieber (1985), although here we shall show that it does not lead to any approximation in parsing.

Precompilation generates multiple instantiations of each rule (1), paired with equivalent CFG rules, stored in a *tuple*:

$$\langle (T_0 \rightarrow T_1 \ldots T_n), rule-name \rangle \quad (4)$$

Each represents the pair-wise unification of some $TFS_1 \ldots TFS_n$ with a sequence of TFSs where each is either lexical, like TFS_l, or derives from some rule instantiation, like TFS_0. For each sequence $T_1 \ldots T_n$ of categories derivable from such instantiations, using the discrimination tree, only the first corresponding instantiation encountered is treated, since all generate the same T_0.

By these means the closure becomes a bounded operation. This approach is valid since slashes are treated separately during parsing, and as in GPSG, Gazdar et. al. (1985), no phrase acceptable on syntactic grounds is then rejected on semantic grounds.

The CFG rules (4) are treated by a parser with a chart containing constituents :

$\langle start, end, T_l, sem_l, slashes_l \rangle$ and

$$\langle start, end, T_0, sem_0, slashes_0 \rangle \quad (5)$$

The latter requires a CFG rule $T_0 \rightarrow T_1 \ldots T_n$ and consecutive constituents:

$$\langle start_1, end_1, T_1, sem_1, slashes_1 \rangle$$
$$\cdots \langle start_n, end_n, T_n, sem_n, slashes_n \rangle$$

where for $j > 1, start_j = end_{j-1}$

Slashes are treated by precompiling a table with entries $\langle T_s, T \rangle$ where T_s represents T_l or T_0, and T is the category assigned to a slash taken from a lexical TFS, and where $retrieve(T_s)$ unifies with $retrieve(T)$ prefixed by $\text{SYNSEM}:\text{LOC}$. Where in (4) $rule-name = filler-head$, (5) is formed only if $\exists \langle T_s, T \rangle \bullet (T_s = T_1 \wedge T = T_{s2} \in slashes_2)$ (6)

This mimics the operation of the $filler-head$ schema. In this case, in (5),

$$slashes_0 = slashes_1 \cup slashes_2 - \{T_{s2}\}$$

whereas with other schemas $\{T_{s2}\} = \varnothing$: this mimics the operation of the *non-local-feature principle*. The CFG grammar generated by this method accepts the same strings as the HPSG grammar and does not just approximate it.

The semantic component sem_0 represents a TFS, identical to sem_j in that sub-constituent known as the *semantic head*, according to the *semantics principle* of HPSG, except that some paths are co-indexed with paths in the semantic components of the other sub-constituents:

$$sem_0 = f(sem_1, \ldots, sem_n) \quad (7)$$

To reduce copying overheads and eliminate TFS unification, sem_0 is encoded in a skeleton form, which is a Prolog structure:

$$s_n(sem_type_0, [p_1, \ldots, p_m]) \quad (8)$$

The type sem_type_0 is not equal to T_0, and is the category of the lexical source of sem_0, which is a *transitive semantic head* of the corresponding TFS_0. For example, the sentence TFS contains a semantic component that derives from the head verb. In parsing, $p_k, 1 \le k \le m$ is bound to the unique identifier of the sub-constituent containing a path that must be co-indexed with a path in sem_j. In the prototype parser this path is the *k*th. one during a traversal of sem_j in the lexical source, that is found to be co-indexed with a syntactic path, that is in turn found to be prefixed by a path co-indexed between the corresponding daughters of the original rule (1) forming TFS_0 (see Section 5).

At retrieval time,

$$sem_0 = sem(sem_type_0, [p_1, \ldots, p_m]) \quad (9)$$

where this indicates the semantic sub-structure in $retrieve(sem_type_0)$, with the addition that each path for co-indexing associated with $1 \le k \le m$ is co-indexed with the appropriate path in the TFS given by $sem(sem_type_{pk}, [p_1, \ldots, p_l])$, (10)

from the p_k th. sub-constituent. A *retrieval graph* is compiled at category allocation time to support *retrieve*. Currently the semantic details of the slash are not recovered: sentences with slashes are accepted identically to ALE.

3 Modification of ALE rules

Figure 1 is the *head-subject-complement* schema in the ALE formalism. Figure 2 shows sections of the single Prolog clause containing 57 goals in 61 lines including the head, produced when ALE compiles the schema. This is invoked by the ALE chart parser after choosing the first of a speculative edge sequence. Its TFS is the structure *SVs*, which resembles (2): the functor represents the type, and each argument is a *value* from a (*feature: value*) pair. The *feature* is retrieved by successive instantiations of an ALE clause compiled from the grammar definition:

```
approps( +type, (-approp_feature:
                    -approp_type))          (10)
```

```
(phrase,Mother,synsem:loc:(cat)
            :(spr:[],subj:[],comps:[]))    M
     ===>
cat> word,HeadDtr,synsem:loc:(cat):
            (head:inv:plus,
             spr:Spr,
             subj:[SubjSynsem],
             comps:CompSynsems)),           D1
goal> (list_sign_to_synsem(CompDtrs,
                      CompSynsems)),        P1
cat>  (SubjDtr),                            D2
goal> (sign_to_synsem(SubjDtr,
                      SubjSynsem)),         P2
cats> (CompDtrs,ne_list),                   D3
goal> head_feature_principle(Mother,
                      HeadDtr),             P3
    semantics_principle(Mother,HeadDtr),    P4
    marking_principle(Mother,HeadDtr),      P5
    nonlocal_feature_principle(Mother,
       HeadDtr,[SubjDtr|CompDtrs])).        P6
```

Figure 1: The Head_subject_complement Schema from HPSG

Goals treat a TFS in a structure *Tag-SVs*, to allow type-coercion during unification into a sub-type, possibly supporting additional (never fewer) features. *Tag*, originally unbound, is then bound to a new copy: argument values are appropriate types (10) for new features, and unchanged for existing ones. Goal 21 references the second edge in the sequence, corresponding to *D2* of the original schema.

```
rule(Tag, SVs, Iqs, Start, End,Edge_no) :-      1
  add_to_type_word(Tag-SVs, Iqs, Iqs_out0),     2
  ud(A-bot, Tag-SVs, Iqs_out0, Iqs_out1),       3
  featval_synsem(Tag-SVs, FS2, Iqs_out1,
                              Iqs_out2),        4
     .    .    .    .    .
  featval_inv(FS5, FS6, Iqs_out5, Iqs_out6),    8
  add_to_type_plus(FS6, Iqs_out6, Iqs_out7),    9
  featval_subj(FS4, FS7,Iqs_out7, Iqs_out8),   10
  featval_hd(FS7, Tag2-SVs2, Iqs_out8,
                              Iqs_out9),       11
  ud(B-bot, Tag2-SVs2, Iqs_out9,
                              Iqs_out10),      12
  featval_tl(FS7,FS8,Iqs_out10, Iqs_out11),    13
  add_to_type_e_list(FS8, Iqs_out11,
                              Iqs_out12),      14
  featval_comps(FS4, Tag3-SVs3, Iqs_out12,
                              Iqs_out13),      15
  ud(C-bot, Tag3-SVs3, Iqs_out13,
                              Iqs_out14),      16
     .    .    .    .    .
  ud(D-bot, E-bot, Iqs_out15, Iqs_out16),      18
  ud(C-bot, F-bot, Iqs_out16, Iqs_out17),      19
  solve(list_sign_to_synsem(E-bot,F-bot),[],
              Iqs_out17, Iqs_out18),           20
  edge(Edge_noB,End,EndB,TagB,SVsB,
              IqsB,DaughtsB,RuleB),            21
     .    .    .    .    .
  deref(I, bot, Tag4, SVsList),                31
  SVsList=..[Type| MemList],                   32
  match_list_rest(Type,MemList,EndB,
       EndEdges,Edge_nos,[],Iqs_out27,
                              Iqs_out28),      33
     .    .    .    .    .
  solve(head_feature_principle(K-bot,L-bot),   46
    [semantics_principle(M-bot,N-bot),         47
     marking_principle(O-bot,P-bot),           48
     nonlocal_feature_principle(Q-bot,R-bot,
       S-bot)], Iqs_out40,Iqs_out41),          49
  add_to_type_phrase(Z-bot, Iqs_out41,
                              Iqs_out42),      50
     .    .    .    .    .
  add_edge_deref(Start,EndEdges,Z,bot,
    Iqs_out52,[Edge_no,Edge_noB|Edge_nos],
    head_subject_complement).                  61
```

Figure 2: The Head-subject-complement Schema after Compilation by ALE

Goal 33 corresponding to *D3* references the remaining edges: their number equals the

number in the *comps* list of the sub-constituent unifying with the head daughter *D1*. Goal 61 creates the edge of the new phrase and corresponds to the mother *M*. Goal 20, and lines 46 to 49 forming one goal, are invocations of ALE procedures, corresponding to *P1*, and *P3* to *P6*. Apart from the first, these lines invoke HPSG principles.

ALE supports inequalities which HPSG does not use, here in lists with names of the form *Iqs_**: the output from one goal is input to the next, and edges contribute new lists for concatenation. The following three goals enforce constraints in the schema:

ud(*Tag1-SVs1, Tag2-SVs2, Iqs_in, Iqs_out*),
featval_FEAT(*FS_in, FS, Iqs_in, Iqs_out*),
add_to_type_TYPE(*FS_in, Iqs_in, Iqs_out*)

The first invokes a general purpose procedure for full-scale unification of the TFSs in its first two arguments. Often this just generates a co-indexing variable for reference elsewhere: *A* in goal 3 corresponds to *HeadDtr* in *D1*. *A* is an unbound *Tag* and *bot* is the most general type in the hierarchy: ud makes *A* reference *Tag-SVs*. If *SVs* becomes subject to type coercion, *A* references the new structure through *Tag*.

The second returns in *FS* the value of FEAT from *FS_in*, type-coercing this when FEAT is not appropriate. The add_to_TYPE goal obtains the common sub-type of TYPE and the type of *FS_in*, which is coerced to adopt this sub-type: the procedure is precompiled from the type hierarchy. Goals 4 to 9 use featval_FEAT and add_to_type_TYPE to enforce a *(path: value)* constraint in the first line of *D1*. Goals 10 to 16 treat three other constraints in *D1*: two of the values are co-indexing variables. For conciseness, the figure omits such goals after the first edge reference. Goals 31 and 32 extract the list of synsem structures *CompDtrs*, returned by the ALE procedure list_sign_to_synsem: the list derives from the value of the list *CompSynsems*.

Goal 50 coerces the initial type of *Z-bot*, the new constituent, to *phrase* as required in *M*. Then unshown goals constrain paths in this TFS according to the *(path: value)* constraints in *M*. Goal 61 invokes a procedure that asserts a new edge containing this coerced TFS referenced by *Z*, between positions *Start*, and *EndEdges* from the last edge of goal 33. The new edge contains a list of edge identifiers in the sequence, and the schema name, from the last two arguments.

The schema of Figure 2 is extended by our precompiler, to generate the tuples in a closure and details of co-indexing in order to guide semantics treatment. The modified goals are shown in Figure 3: clause modification is easier than modifying the complex compiler code in ALE, and the compiled schema already invokes ALE procedures appropriately.

During the closure, procedures invoked by goals 21 and 33 must constrain rule application so each sequence of edges is treated just once by each rule. Prolog backtracking cannot be altered to achieve this, and the edge and match_list_rest goals are modified. A list of identifiers of edges already invoked is passed between instances of these goals.

Detection of co-indexing requires access to the TFS in each sub-constituent after constraint application, and to the new TFS inside add_edge_deref before edge assertion, when co-indexing information is lost in copying. Since each schema is applied without backtracking to a single sequence of edges each containing *Tag-Bot*, the list of sub-constituent TFSs can be returned through the head of the rule: the new TFS, *Z-bot* appears as two arguments of the last goal.

```
rulejcb(Tag,SVs,Iqs,Start,End,Edge_no,
   - [Tag-SVs,TagB-SVsB,MemList],
   - Z-bot, + Edge_countA,
   - head_subject_complement, + Edges_in,
   -Edges_outC):-                                1
edgejcb(Edge_noB,StartB,EndB,TagB,SVsB,
      IqsB,DaughtsB,RuleB,Edge_countA,
      Edge_countB, Edges_in, Edges_outB),  21
match_list_restjcb(Type,MemList,EndB,
      EndEdges,Edge_nos,[],Iqs_out27,
      Iqs_out28,Edge_countB, Edge_countC,
      Edges_outB, Edges_outC),              33
replace_add_edge_deref(Start, EndEdges,
                        Z, bot, Iqs_out52,
          [Edge_no, Edge_noB | Edge_nos],
           head_subject_complement )        61
```

Figure 3: The Head_subject_complement Schema after Further Compilation

The extra argument 7 of rulejcb is a list of sub-constituent TFSs after constraint application. *Z-bot* is the new TFS. *Edge_countA* is an initial count of 1 of the edges encountered so far. Remaining arguments are the rule name, and lists of edge identifiers before and after rule application.

Goal 21 invokes a new clause that invokes edge, and adds 1 to *Edge_countA* to form *Edge_countB*. The edge number, *Edge_noB*, is added to the head of *Edges_in*, to form *Edges_out*. Goal 33 invokes a recursive clause which similarly treats elements of *MemList*.

Goal 61 invokes a new procedure without additional arguments, to assign T_0 to the new TFS, *Z-bot*, using the discrimination tree and to assert a tuple (4). If T_0 is new, a fully dereferenced *Z-bot* is added to the retrieval graph, and a new edge asserted with the retrieved TFS. Another asserted clause associates T_0 with the edge number: similar clauses were asserted in lexical edge creation. They are referenced in tuple formation, from edge numbers in argument 6.

For debugging, an asserted clause contains the string deriving a tuple, structured into sub-phrases using brackets. Only the first sub-phrase deriving each category is used, to restrict numbers. Even so, this permitted the detection of over-generation arising from ungrammatical strings. This arose from the verbs *is, can, be, seem,* and the infinitival *to* not specifying their subject beyond co-indexing it with the subject of their complement, which is variously another (sometimes infinitival) verb or a predicate. This allowed generation of infinite sequences like "*X is is*…" where *X* is any phrase. It was cured by hand-specifying each subject as `np`. Complements were similarly treated for *believe* and *expect*. An automatic approach would propagate possible subjects, including alternatives to `np` in a larger grammar, from the complement into the outer verb.

To allow for large numbers of phrasal edges, edges optionally have unbound *SVs* arguments which are bound using the retrieval graph when referenced With over-generation cured, only 18,000 tuples were generated, and the facility was unused.

4 The Closure Algorithm

Using the TFS of each lexeme, an edge is asserted with an identifier between *Bottom* and *Last_free_edge_number-1*: *Start* and *End* are unbound so every sequence of edges is considered by rulejcb. Then the algorithm of Figure 4 is executed. On each recursive invocation, numbered in argument three, repeat_apply makes a pass over these edges and new edges generated in previous invocations. The first unused identifier after a pass is asserted in last_free_edge_number, and when this is unchanged after a pass in which no edges were asserted, termination occurs.

The first edge in a sequence is selected by apply_schemas_to_a_first_edge: *N* is the identifier which is incremented on each recursion between *Bottom* and *End*. If *SVs* in an edge is unbound, it is re-formed from the retrieval graph by add_SVs_to_edge.

Selection of the remaining edges in a sequence occurs non-deterministically within the compiled grammar rules. These are invoked by try_all_rules which references rulenames, previously asserted to identify all rules in a list. It invokes try_all_rules/4 to recurse through the list, each time invoking try_all_edges inside a negation, since a failure-driven loop treats multiple edge sequences by invoking the non-deterministic rulejcb. The TFS of the first edge appears in the first two arguments: its identifier *N* appears alone and as the first in a list of edge identifiers for the sequence, completed inside rulejcb.

To minimise compilation time, each sequence must be treated once by each rule. Sequences unpredictably contain 2 or 3 edges, and a first edge *E1* can combine with edges created by sequences of edges treated after *E1*. No edge can ever be discarded as a candidate for any daughter in any rule. Sequences are too numerous to record by asserting clauses to be referenced before rule application.

The chosen solution is straightforward and fairly efficient. On each pass of repeat_apply, a range of acceptable edge identifiers is established. Four global variables holding identifiers, *Bottom*, *Top*, *End* and *Last_free_edge_number* are asserted in clauses bottom, top, end and last_free_edge_number

respectively. The first three are adjusted in repeat_apply, whilst the last is incremented in replace_add_edge_deref on edge assertion.

On any pass, edges with *N>End* are asserted, and apply_schemas_to_a_first_edge treats first edges between *Bottom* and *End*. At the end of a pass, *Top* is set equal to *End* and *End* is then set to *Last_free_edge_number-1*: before the first pass *Top is set to Bottom – 1.*

```
repeat_apply(Bottom,Last_free_edge_number,
              Pass_no):-
  end(End),
  apply_schemas_to_a_first_edge(Bottom,End),
  last_free_edge_number(Last_free_edge_no2),
  End2 is Last_free_edge_number2 – 1,
  retractall(end(_)),assert(end(End2)),
  Pass_no_out is Pass_no + 1,
  (! , ( (not(Last_free_edge_number = =
                Last_free_edge_number2))
            ->
     (repeat_apply(Bottom,Last_free_edge_no2,
          Pass_no_out),!)
            ;true
   ).
apply_schemas_to_a_first_edge(N,L):-
  edge(N,Start,End,Tag, SVs, Iqs, Dtrs,
       Rule_name),!,
  (var(SVs) -> add_SVs_to_edge(N,Tag,SVs)
             ; true),
  try_all_rules(Tag,SVs,N),!,
  New_N is N + 1,
  (not(New_N > L) ->
  (apply_schemas_to_a_first_edge(New_N,L))
                     ; true).
try_all_rules(Tag,SVs,N):-
  rulenames(Rulenames),
  try_all_rules(Tag, SVs,N,Rulenames).
try_all_rules(Tag,SVs,N,
               [Rulename|Rulenames]):-
  not(try_all_edges(Tag, SVs,N,Rulename)),
  try_all_rules(Tag, SVs,N,Rulenames),!.
try_all_edges(Tag, SVs,N,Rulename):-
  rulejcb(Tag, SVs, [], _ , _ , N, Daughters,
          Mother,1,Rulename,[N],D2),
  fail.
```

Figure 4: Algorithm for Tuple Generation

Consequently, edges between *Top+1* and *End* were always created during the last pass: initially this is the set of lexical edges.

In edgejcb and match_list_restjcb, an edge identified by *d2* or *d3* depending on position in a sequence, has its *SVs* unified with the rule daughter if a following test succeeds. *Test 1* succeeds if *d1* was created on the previous pass (or as a lexical edge, for pass 1), and *d2* (and *d3* for a 3-daughter sequence), were created on any pass (including lexemes) up to and including the last. The *d1* restriction prevents treatment on multiple passes. If *Test 1* fails, *Test 2* succeeds if *d2* is newly created on the previous pass, and *d1*, (and *d3*), were created on any pass (including lexemes) up to and including the last. If this fails, *Test 3* is passed if *d1* and *d2* were created on any pass up to but not including the last pass, and if the same rule successfully treated them earlier as the start of a 3-edge sequence.

By delaying combination of a new edge into a sequence, until the pass after its creation, we avoid a repeat pass to catch the case where the other edges do not yet exist at creation time. Overall this necessitates < 1 extra, low cost pass. HPSG eliminates sequences mainly by constraints between daughters, avoided on this extra pass by trivial arithmetic comparisons in $O\left(\left(End - Top\right) \times \left(End - Bottom\right)\right)$ whilst the unavoidable $O\left(End - Top\right)$ costs of first-daughter unification are small

Test 3 treats a new *d3*, with old *d1* and *d2*, one of which must have been new on an earlier pass, when it was treated under *Test 1* or *2*. Since *d3* is a potential complement of either *d1* or *d2*, some sequence *[d1, d2, d3']* was treated earlier, even if *d3'* failed unification: at that time treated_edges_before(*Rulename, [d1,d2])* was asserted. Unification of *d2* in *[d1, d2, d3]* takes place (on repeated passes) only if this asserted edge is found. Successful unifications will be repeated, but the closure on our test grammar has only 7 passes, and in backtracking only one *[d1, d2]* unification takes place for all *[d1, d2, d3]*.

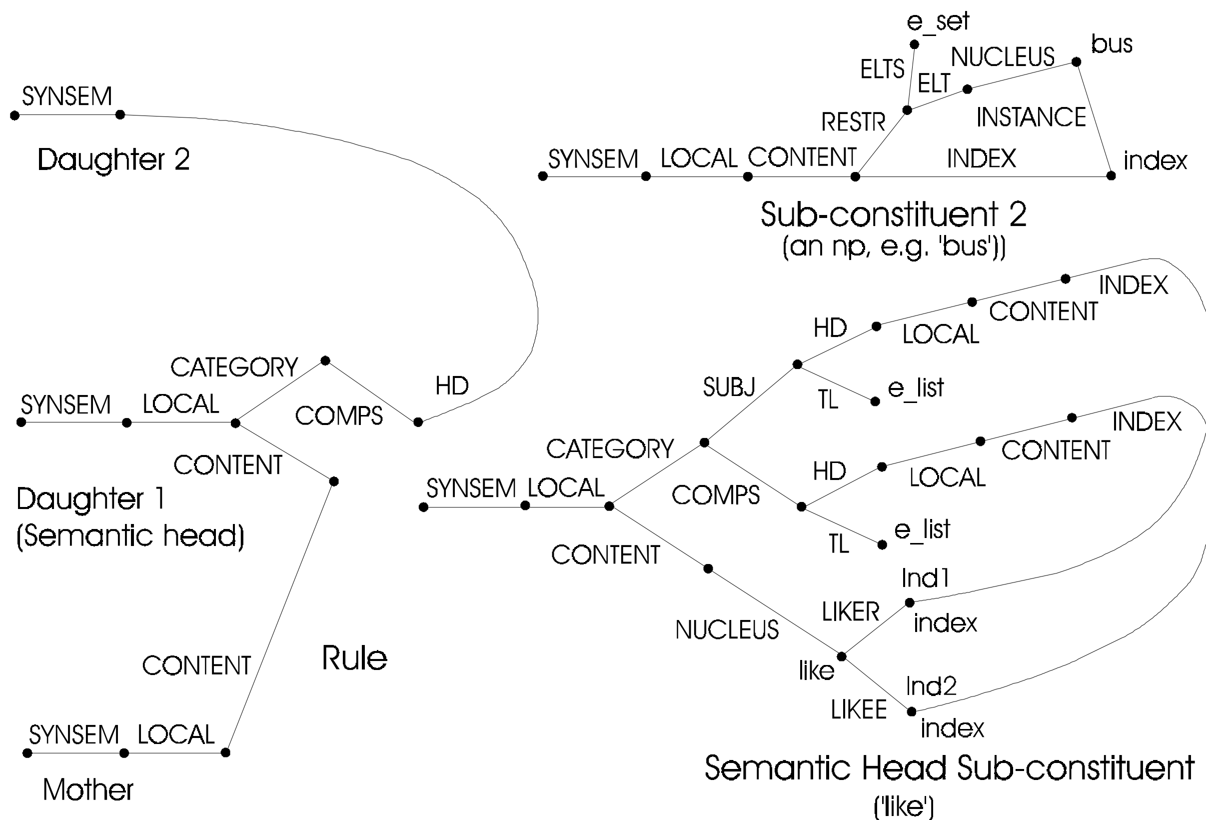

Figure 5: Application of Head-Complement Schema to an Edge Sequence

5 Treatment of Semantic Structures

In a constituent, the co-indexing of paths in a copy of the semantic sub-structure of a semantic head was explained in (7) to (10). Figure 5 illustrates this for the *head-complement* schema.. In the verb (the semantic head), the semantic and syntactic paths *Pm* and *Ps* co-index at the variable *Ind2*, where :
Pm = SYNSEM: LOCAL: CONTENT: NUCLEUS: LIKEE and
Ps = SYNSEM: LOCAL: CATEGORY: COMPS: HD: LOCAL: CONTENT: INDEX
The rule co-indexes a pair of syntactic paths in head and non-head daughters:
Ph = SYNSEM: LOCAL: CATEGORY: COMPS: HD and
Pnh = SYNSEM

Since the concatenation *(Ph : Psuffix) = Ps* where *Psuffix* = LOCAL: CONTENT: INDEX, then the TFS reached by *Pm* in the head unifies with that reached by *(Pnh: Psuffix)* in the other sub-constituent. The *index* structure reached by INDEX has a dual role in an np: as well as appearing in the semantic structure of the phrase, it also tests syntactic agreement of its *number*, *person* and *gender* features via co-indexing in the rule. It was therefore considered in category allocation.

Analysis involves applying each grammar rule to a sequence of *Tag-bot* structures. In these, the *Tag* of each co-indexed node is bound to a 3-digit integer *xyy*, where *yy* and *x* are ordinals identifying the co-indexed node in the rule, and the first daughter with a path to that node. The head daughter is distinguished since its path SYNSEM: LOCAL: CONTENT co-indexes with that path in the mother. This leads to <*Ph, Pnh*> pairs, each identified by an integer *Path_no*, for each rule. For each pair in each rule a clause is asserted: rule_paths(*Rulename, Path_no, _, Daught_no*) where *Daught_no* locates *Pnh*.

Each lexeme that a tuple (4) shows to be unifiable with the head daughter of some rule, is treated to detect 3-tuples of the form:
< *Indn, Path_no, Suffix_no* > according to the mechanism illustrated in the example above: *Suffix_no* identifies a single *Psuffix* path. A skeleton semantic structure (8) is derived,

where each p_k argument corresponds to a 3-tuple, ordered as *Indn* nodes are encountered in a TFS traversal. *Path_no* fields appear in the same order in an asserted clause r_n(*cat, Paths*) where *cat* is T_l allocated to the lexeme. During constituent construction in CFG parsing, *Daught_no* of the sub-constituent is known, and *Rulename* is deduced from the tuple (4), so r_n and rule_paths identify the p_k argument in (8) to bind to the edge identifier of the sub-constituent.

Retrieval graph arcs leading to nodes like *Ind2* are marked by asserted clauses:

arc_to_retrieval_tag2(+*Current_arc_no,*
+*Category, _ , -Path_no, -Suffix_no*).

For each possible <*Path_no, Suffix_no*> pair generating *(Pnh : Psuffix)*, this path is speculatively followed in the retrieval graph for each lexeme to identify a node and assert:

find_type_node_compiled(+*Path_no,*
+*Category,* +*Suffix_no,*
-Type_node_no, -Arc_no_in).

When retrieving the TFS_0 of a CFG constituent from the retrieval graph, once the semantic path *semantics* from (3) is encountered, the sem_type_0 of (8) is treated, starting at the node reachable by *semantics*. Where an arc number matches that in an arc_to_retrieval_tag2 clause, *Path_no* and *Suffix_no* are used to address find_type_node_compiled. *Category* here derives from sem_type_{pk} from (10), being the category of the semantic skeleton in a sub-constituent identified by the appropriate p_k in (8): the ordering ensures that successive arcs matching arc_to_retrieval_tag2 clauses correspond correctly with consecutive p_k arguments. The arc in the TFS being constructed is redirected to a copy of the indicated node, and traversal of the retrieval graph continues from that node. *Arc_no_in* is used recursively with arc_to_retrieval_tag2 to detect if the target type-node should itself be replaced by a node derived from a further sub-constituent.

This technique also properly treats the *head-subject-complement*, *subject-head* and *adjunct-head* rules. In this last case the RESTR set of identifiers for the noun and adjectives is properly constructed. This set is not accessible from the semantic TFS of the sentence since the grammar (probably incorrectly) co-indexes the *Indn* of the verb with the node reached by INDEX in the `np`, rather than with that from which the INDEX and RESTR arcs emerge. The *specifier-head* rule uses a different form of co-indexing which we have not yet treated: appropriate semantic structures are still returned for the sample grammar, where no specifier has a more specific *index* structure than any head `np`. The counter-example *"a sheep*" which derives its *number* from the specifier is not in the lexicon. Similarly, the approximation that an arc in the head is redirected to a node in another sub-constituent avoids unification of *index* structures to properly treat "… *sheep eat(s)":* such low-cost unification can easily be added to retrieval.

Our precompiled CFG is an exact equivalent of the TFS grammar rather than an approximation, since our restrictor does not eliminate paths affecting agreement except for *slash*: slashes are treated separately in our parsing algorithm, as in CFG. Since the schemas treated enforce agreement through a syntactic path in the head, the semantics in the head can be omitted by the restrictor. This also eliminates the major source of TFS expansion in a closure. The test grammar does not make the daughter TFSs of a phrase accessible except through its semantics, so these are also eliminated from consideration.

The path to the *index* structure in an `np` is co-indexed by the mechanism in Figure 5, and by other schemas, some of which co-index it with *index* in another sign combining with `np` to produce `np`. Therefore it is re-introduced for category allocation after semantics is excluded (3). The RESTR component is still excluded: a linguistic reason is that its value depends on word morphology, whilst syntactic agreement depends on more general features.

However, no automatic mechanism could restrict the *index* treatment as we do. Verbs like *believe*, *seem*, *persuade*, *expect* and *promise* take a `vp` or an `s` as complement. They could potentially specify agreement with the semantic part of a `vp` of varying saturation. In practice only the syntactic HEAD of the

complement is constrained, so these verbs differ from verbs that take `np` as a subject. A linguistic reason is that the semantic structure of a verb depends on its word morphology as in Figure 5, whilst in an `np` this dependency applies only to RESTR and not to INDEX.

An automatic mechanism to generate our restrictor would necessitate a closure from a sample of the lexicon, since only when syntactic agreement occurs can the need for semantic agreement be assessed. The linguist can predict such agreement by inspection, so a better approach might be to automatically generate diagrams like Figure 5 to guide in the choice of restrictor. An over-drastic restrictor becomes apparent only when a retrieved TFS from CF parsing does not match the original from TFS parsing. Automatic mechanisms for comparison might be worth investigating.

Automatic mechanisms to derive our treatment of slashes may be possible, since they are associated with lists that grow during parsing, not shrink as do *subcat*, *subj* or *comps* lists. Our test grammar does not maintain quantifier lists, but their behaviour is in many ways similar to that of slashes.

We do not currently retrieve the semantic structure of slashes. If each slash is paired with the edge number of its lexical origin, and details of slashes satisfying the *filler-head* schema during parsing are indexed by edge number, then the semantics can be retrieved from the TFS corresponding to T_1 in (6), when the slash is encountered during retrieval.

6 Conclusion

It has proved practical to precompile in an acceptable time a realistic HPSG grammar into exactly equivalent (neglecting semantics) CFG categories and rules, of reasonable number and compact size, together with a table to control slash agreement. It was also possible to generate data structures for building skeleton semantic structure and retrieving its full structure after parsing, obviating the need for TFS unification. A Prolog prototype parses 18 times faster, and is estimated to be 180 times faster in an optimum imperative code solution. This predicted speed-up would exceed that obtained with a CFG approximation, where TFS unification must follow CFG parsing. The kind of co-indexing used in the *specifier-head* schema is not treated in semantic retrieval, but the method seems extensible to embrace this.

Acknowledgements

This research was funded by a legacy from Miss Nora Brown, and Workshop attendance funded by INVU. Our thanks to the anonymous referees and to Mr. Stephan Oepen for their suggestions, and to Bernd Kiefer and Kentaro Torisawa for copies of their papers.

References

Brown, J.C. and Manandhar, S. (2000) *Compilation versus Abstract Machines for Fast Parsing of Typed Feature Structure Grammars*, Future Generation Computer Systems 16, pp. 771-791.

Carpenter, B. and Penn, G. (1996) *Compiling Typed Attribute-value Logic Grammars*, in “Recent Advances in Parsing Technology”, H.Bunt, M. Tomita, ed., Kluwer Academic Press, Dordrecht, pp. 145-168.

Carroll, J.A. (1993) *Practical Unification-based Parsing of Natural Language*, Technical Report No. 314, University of Cambridge Computer Laboratory.

Gazdar, G., Klein, E., Pullum, G., Sag, I. (1985) *Generalized Phrase Structure Grammar,* Blackwell, Oxford.

Kiefer, B. and Krieger, H.-U. (2000) *A Context-Free Approximation of Head-driven Phrase Structure Grammar*, in Proceedings of the 6th. Int. Workshop on Parsing Technologies (IWPT), Trento, Italy, pp. 135-146.

Matheson, C. (1996) *Developing HPSG Grammars in ALE*, Course Notes, Human Communications Research Centre, University of Edinburgh. http://www.ltg.hcrc.ed.ac.uk/projects/ledtools/ale-hpsg/index.html.

Pollard, C. and Sag, I.A. (1994) *Head-Driven Phrase Structure Grammar*, University of Chicago Press, Chicago.

Shieber, S.C. (1985) *Using Restriction to Extend Parsing Algorithms for Complex Feature Based Formalisms*, in Proceedings of the 23rd. Annual Meeting of the Association for Computational Linguistics, pp. 145-152.

Torisawa, K., Nishida, K., Miyao,Y., and Tsujii, J.-I. (2000) *An HPSG Parser with CFG Filtering*, Natural Language Engineering 6 (2), pp. 1-18.