Packing of Feature Structures for Efficient Unification of Disjunctive Feature Structures

Yusuke Miyao
Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan
E-mail: yusuke@is.s.u-tokyo.ac.jp

Abstract

This paper proposes a method for packing feature structures, which automatically collapses equivalent parts of lexical/phrasal feature structures of HPSG into a single *packed feature structure*. This method avoids redundant repetition of unification of those parts. Preliminary experiments show that this method can significantly improve a unification speed in parsing.

1 Introduction

Efficient treatment of syntactic/semantic ambiguity is a key to making efficient parsers for wide-coverage grammars. In feature-structure-based grammars[1], such as HPSG (Pollard and Sag, 1994), ambiguity is expressed not only by manually-tailored disjunctive feature structures, but also by enumerating non-disjunctive feature structures. In addition, there is ambiguity caused by non-determinism when applying lexical/grammar rules. As a result, a large number of lexical/phrasal feature structures are required to express ambiguous syntactic/semantic structures. Without efficient processing of these feature structures, a sufficient parsing speed is unattainable.

This paper proposes a method for packing feature structures, which is an automatic optimization method for parsers based on feature structure unification. This method automatically extracts equivalent parts of feature structures and collapses them into a single *packed feature structure*. A packed feature structure can be processed more efficiently because we can avoid redundant repetition of unification of the equivalent parts of original feature structures.

There have been many studies on efficient unification of disjunctive feature structures (Kasper and Rounds, 1986; Hasida, 1986; Dörre and Eisele, 1990; Nakano, 1991; Blache, 1997; Blache, 1998). All of them suppose that disjunctive feature structures should be given by grammar writers or lexicographers. However, it is not practical to specify all ambiguity using only manually-tailored disjunctive feature structures in grammar development. Where disjunctive feature structures cannot be given explicitly those algorithms lose their advantages. Hence, an automatic conversion method, such as the packing method described hereafter, is required for further optimization of those systems. In addition, this packing method converts general feature structures to a suitable form for a simple and efficient unification algorithm which is also described in this paper.

Griffith (Griffith, 1995; Griffith, 1996) points out the same problem and proposes a compilation method for feature structures called *modularization*. However, modularization is very time-consuming, and is not suitable for optimizing feature structures produced during parsing. An earlier paper of myself (Miyao et al., 1998) also discusses the same problem and proposes another packing method. However, that method can pack only pre-specified parts of input feature structures, and this characteristic limits the overall efficient gain. The new method in this paper can pack any kind of feature structures as far as possible, and is more general than the previous method.

2 Data Structure and Algorithms

This section describes the data structure of packed feature structures, and the algorithms for packing and unification of packed feature structures. Through of this section, I will refer to examples from the XHPSG system (Tateisi

[1] In this paper we consider *typed feature structures* described in (Carpenter, 1992).

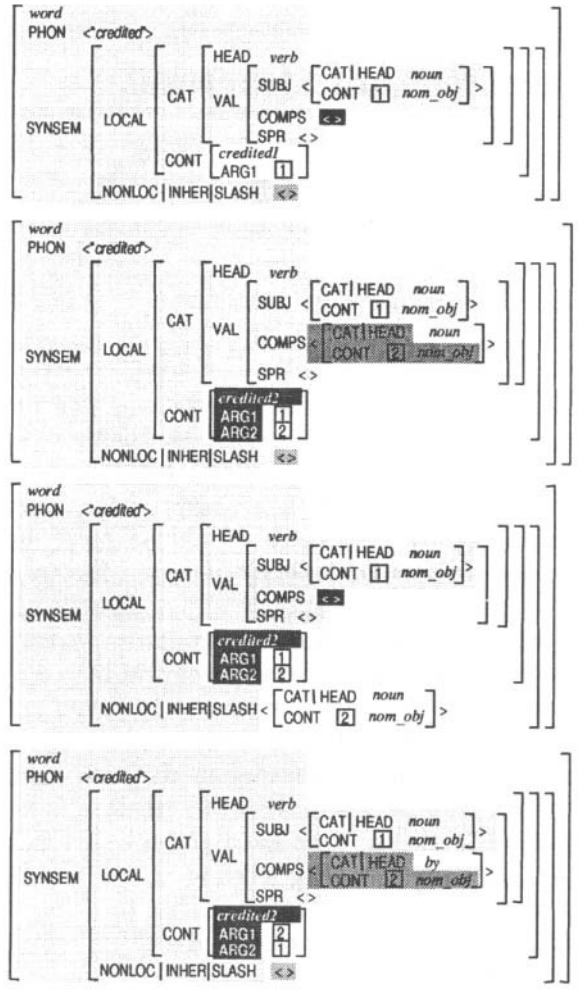

Figure 1: 4 out of 37 lexical entries which the XHPSG system assigns to the word "*credited*". Parts shaded with the same pattern are equivalent.

et al., 1998), an HPSG-based grammar for English.

2.1 Packed Feature Structure

Figure 1 shows 4 out of 37 lexical entries which the XHPSG system assigns to the word "*credited*". These lexical entries have various equivalent parts in their respective feature structures. In Figure 1, equivalent parts are shaded with the same pattern.

Figure 2 shows a packed feature structure for the lexical entries shown in Figure 1. Note that the equivalent parts of the original feature structures are collapsed into a *feature structure segment*, which is denoted by S_i in Figure 2. S_0 is a special segment called the *root segment*, which

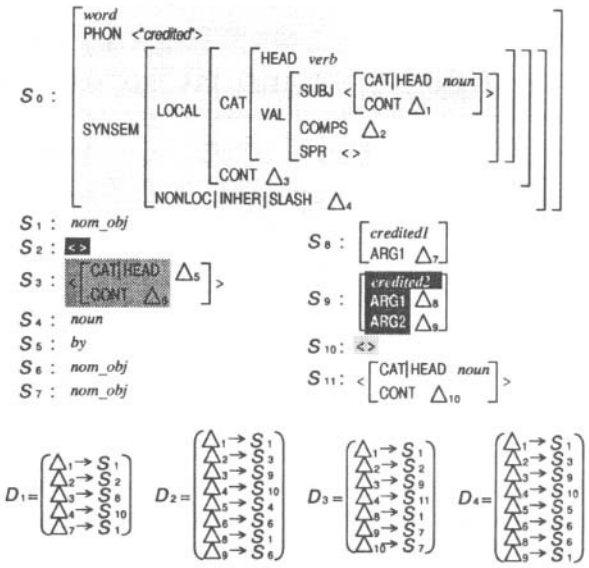

Figure 2: A packed feature structure expressing the same information as the set of feature structures in Figure 1. Shaded parts correspond to the parts with the same pattern in Figure 1.

describes the root nodes of all original feature structures. Each segment can have *disjunctive nodes*, which are denoted by Δ_i. For example, S_3 has two disjunctive nodes, Δ_5 and Δ_6. A *dependency function*, denoted by D_i, is a mapping from a disjunctive node to a segment, and each D_i corresponds to one original feature structure. We can obtain each original feature structure by replacing each disjunctive node with the output of the respective dependency function.

For applying the unification algorithm described in Section 2.3, we introduce a condition on segments: *a segment cannot have inter- or intra-segment shared nodes.* For example, the disjunctive node Δ_1 in Figure 2 must be introduced for satisfying this condition, even though the value of this node is the same in all the original feature structures. This is because this path is structure-shared with another path (`SYNSEM|LOCAL|CONT|ARG1` and `SYNSEM|LOCAL|CONT|ARG2`). Structure-sharing in original feature structures is instead expressed by letting the dependency function return the same value for different inputs. For example, result values of applying D_1 to Δ_1 and Δ_7 are both S_1.

The reason why we introduce this condition is to guarantee that a disjunctive node in the

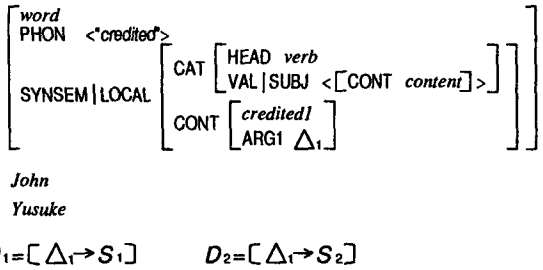

Figure 3: A sample packed feature structure. If it is unified with the top feature structure in Figure 1, a new disjunctive node must be introduced to **SYNSEM|LOCAL|CAT|VAL|SUBJ|FIRST|CONT**.

result of unification will appear only at a path where a disjunctive node appears in either of the input feature structures at the same path. For example, suppose we unify the top feature structure in Figure 1 with the packed feature structure in Figure 3. In the result of unification, a new disjunctive node must appear at **SYNSEM|LOCAL|CAT|VAL|SUBJ|FIRST|CONT**, while no disjunctive nodes appear in either of the input feature structures at this path. By introducing such a disjunctive node in advance, we can simplify the algorithm for unification described in Section 2.3.

Below I first describe the algorithm for packing feature structures, and then the algorithm for unification of packed feature structures.

2.2 Algorithm for Packing

The procedure `pack_feature_structures` in Figure 4 describes the algorithm for packing two packed feature structures, denoted by $\langle \mathcal{S}', \mathcal{D}' \rangle$ and $\langle \mathcal{S}'', \mathcal{D}'' \rangle$. $\mathcal{S}'$ and $\mathcal{S}''$ denote sets of segments, and $\mathcal{D}'$ and $\mathcal{D}''$ denote sets of dependency functions. We start from comparing the types of the root nodes of both feature structures. If either of the nodes is a disjunctive node (*Case 1*), we compare the type of the other feature structure with the type of each disjunct, and recursively pack nodes with the same type if they exist (*Case 1.1*). Otherwise, we just add the other feature structure to the disjunctive node as a new disjunct (*Case 1.2*). If the types of the nodes are equivalent (*Case 2*), we collapse them into one node, and apply packing recursively to all of their subnodes. If they are not equivalent (*Case 3*), we create a new disjunctive node at this node, and let each original

```
procedure pack_feature_structures(⟨S', D'⟩, ⟨S'', D''⟩)
begin
  S := φ
  S'_0 ∈ S', S''_0 ∈ S''
  pack(S'_0, S''_0)
  D := D' ∪ D''
  return ⟨S, D⟩
end

procedure pack(F', F'')
begin
  if F' (or F'') is disjunction then                    ... Case 1
    if ∃G(G ∈ disjuncts(F'),
          G and F'' have equivalent types) then         ... Case 1.1
      S := S∪disjuncts(F')
      pack(G, F'')
      D'' := {D|D'' ∈ D'', D = D'' ∪ (F' → F'')}
    else                                                ... Case 1.2
      S := S∪disjuncts(F')∪{F''}
      D'' := {D|D'' ∈ D'', D = D'' ∪ (F' → F'')}
    endif
  else if F' and F'' have equivalent types then         ... Case 2
    F' := F''
    foreach f in features(F')
      pack(follow(f, F'), follow(f, F''))
  else                                                  ... Case 3
    S := S ∪ {F', F''}
    F := disjunctive_node
    D' := {D|D' ∈ D', D = D' ∪ (F → F')}
    D'' := {D|D'' ∈ D'', D = D'' ∪ (F → F'')}
  endif
end

disjuncts:  return a set of disjuncts of the disjunctive node
features:   return a set of features
follow:     return a substructure reached by the specified feature
```

Figure 4: Algorithm for packing two packed feature structures $\langle \mathcal{S}', \mathcal{D}' \rangle$ and $\langle \mathcal{S}'', \mathcal{D}'' \rangle$.

feature structure from this node become a new segment.

For simplicity, Figure 4 omits the algorithm for introducing disjunctive nodes into shared nodes. We can easily create disjunctive nodes in such places by preprocessing input feature structures in the following way. First each input feature structure is converted to a packed feature structure in advance by converting shared nodes to disjunctive nodes. Then the above algorithm can be applied to these converted packed feature structures.

2.3 Algorithm for Unification

Below I describe the algorithm for unification of packed feature structures, referring to the example in Figure 2. Suppose that we are unifying this packed feature structure with the feature structure in Figure 5. This example considers unification of a non-packed feature structure with a packed feature structure, although this algorithm is capable of unifying two packed feature structures.

The process itself is described by the procedure `unify_packed_feature_structures` in Figure 6. It is quite similar to a normal uni-

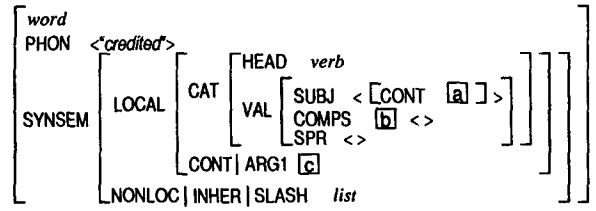

Figure 5: A sample feature structure to be unified with the packed feature structure in Figure 2.

```
procedure unify_packed_feature_structures(⟨S', D'⟩,⟨S'', D''⟩)
begin
  S := φ, D := φ
  foreach D' ∈ D' and D'' ∈ D''
NEXT:
  begin
    push_segment_stack(S'_0 ∈ S', S''_0 ∈ S'')
    do until segment_stack_is_empty
    begin
      pop_segment_stack(S', S'')
      if S' is disjunction then S' := D'(S')                    ... (1)
      if S'' is disjunction then S'' := D''(S'')
SEGMENT_UNIFY:
      if already_unified(S', S'') then                          ... (2)
        S := restore_unify_result( S', S'')
        S' := S, S'' := S                                       ... (3)
      else
        if S := unify(S', S'') fails then
          goto NEXT
        else
          S := S ∪ {S}
          set_unification_result(S, S', S'')
          S' := S, S'' := S                                     ... (4)
        endif
      endif
    end
    D := D ∪ {D' ∪ D''}
  end
  return ⟨S, D⟩
end

procedure unify(F', F'')
begin
  if F' or F'' is disjunction then                              ... (5)
    F := disjunctive_node
    push_segment_stack(F', F'')
  else
NODE_UNIFY:
    F := unify_type(F', F'')
    foreach f in features(F)
      follow(f, F) := unify(follow(f, F'), follow(f, F''))
  endif
  return F
end

already_unified:  true when unification is already computed
restore_unify_result:  restore the result of unification from
                       the table
set_unify_result:  store the result of unification into the table
unify_type:  return the unification of both types
```

Figure 6: Algorithm for unifying two packed feature structures $\langle S', D' \rangle$ and $\langle S'', D'' \rangle$.

fication algorithm. The only difference is the part that handles disjunctive nodes. When we reach a disjunctive node, we put it onto a stack (`segment_stack`), and postpone further unification from this node ((5) in Figure 6). In this example, we put Δ_1, Δ_2, Δ_3, and Δ_4 onto the stack. At the end of the entire unification, we

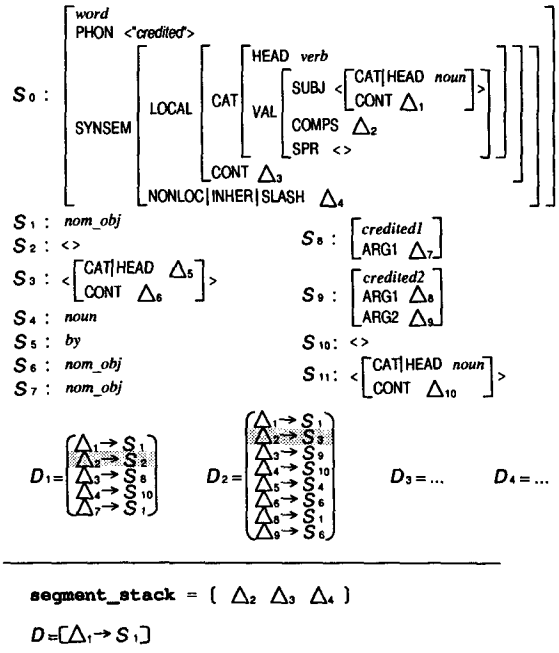

Figure 7: Intermediate data structure after unifying Δ_1 with [a]. Disjunction is expressed by non-determinism when applying the dependency functions. When we unify a feature structure segment for Δ_2, we unify S_2 if we are applying D_1, or S_3 if D_2.

apply a dependency function to each member of the stack, and unify every resulting segment with a corresponding part of the other feature structure ((1) in Figure 6). In this example, we apply D_1 to Δ_1, which returns segment S_1. We therefore unify S_1 with the feature structure tagged as [a] in Figure 5.

Disjunction is expressed by non-determinism when applying the dependency functions. Figure 7 shows the intermediate data structure after unifying Δ_1 with [a]. We are now focusing on the disjunctive node Δ_2 which is now on the top of `segment_stack`. When we are applying D_1, we unify S_2 with the corresponding feature structure [b]. Should we instead apply D_2, S_3 would be unified.

A benefit of this unification algorithm is that we can skip unification of feature structure segments whose unification is already computed ((2) in Figure 6). For example, we unify segment S_0 with the other feature structure only once. We can also skip unification of S_1 and S_{10} for D_2, because the result is already computed

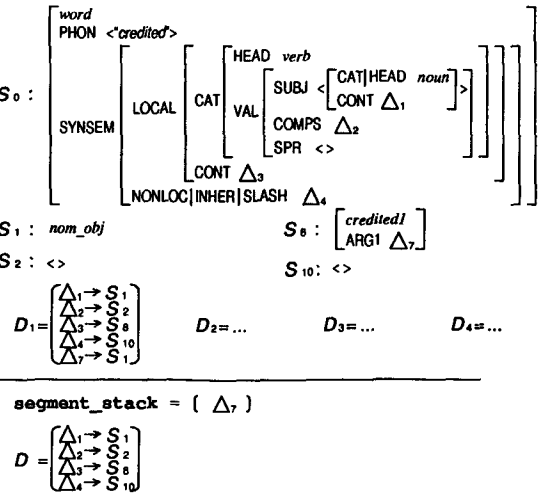

Figure 8: Intermediate data structure after the unification of Δ_4. Because the result of applying D_1 to Δ_7 is already overwritten by the result of unifying S_1 with [a], we unify this resulting feature structure with [c].

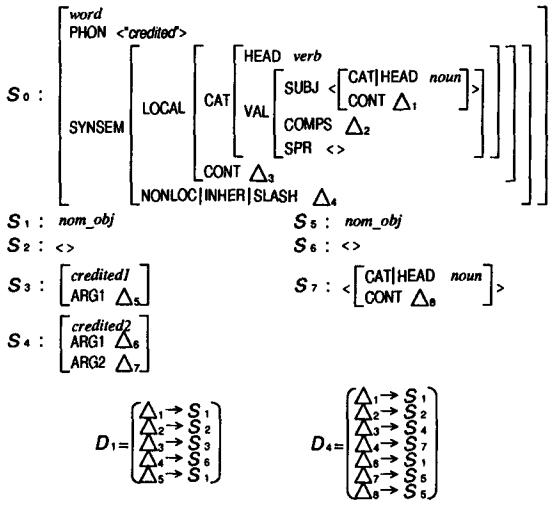

Figure 9: The resulting packed feature structure of unifying the packed feature structure of Figure 2 with the feature structure of Figure 5.

for D_1. This operation preserves the validity of unification because each segment does not have inter- or intra-segment shared nodes, because of the condition we previously introduced.

Note that this method can correctly unify feature structures with reentrancies. For example, Figure 8 shows the intermediate data structure after unifying Δ_4, and the process currently reached Δ_7 and [c]. The result of the application of D_1 to Δ_7 is the result of unifying S_1 with [a], because S_1 is overwritten with the result of this previous unification ((3) and (4) in Figure 6). Hence, we unify [c] with this result.

Above unification algorithm is applied to every combination of dependency functions. The result of the entire unification is shown in Figure 9.

3 Experiments

I implemented the algorithms for packing and unification in LiLFeS (Makino et al., 1998). LiLFeS is one of the fastest inference engines for processing feature structure logic, and efficient parsers have already been realized using this system. For performance evaluation I measure the execution time for a part of application of grammar rules (i.e. schemata) of XHPSG.

Table 1 shows the execution time for unifying the resulting feature structure of applying schemata to lexical entries of "*Mary*" as a left daughter, with lexical entries of "*credited*"/"*walked*" as right daughters. Unification of packed feature structures achieved a speed-up by a factor of 6.4 to 8.4, compared to the naive approach. Table 2 shows the number of unification routine calls. `NODE_UNIFY` shows the number of nodes for which unification of types is computed. As can be seen, it is significantly reduced. On the other hand, `SEGMENT_UNIFY` shows the number of check operations whether unification is already computed. It shows that the number of node unification operations is significantly reduced by the packing method, and segment unification operations account for most of the time taken by the unification.

These results indicate that a unification speed can be improved furthermore by reducing the number of the segment unification. The data structure of dependency functions has to be improved, and dependency functions can be packed. I observed that at least a quarter of the segment unification operations can be suppressed. This is one of the future works.

4 Conclusion

The packing method I described in this paper automatically extracts equivalent parts from feature structures and collapses them into a single packed feature structure. It reduces redundant repetition of unification operations on the

Table 1: Execution time for unification. *Test data* shows the word used for the experiment. *# of LEs* shows the number of lexical entries assigned to the word. *Naive* shows the time for unification with a naive method. *PFS* shows the time for unification of packed feature structures (PFS). *Improvement* shows the ratio *(Naive)/(PFS)*.

Test data	*# of LEs*	*Naive* (msec.)	*PFS* (msec.)	*Improvement* (factor)
credited	37	36.5	5.7	6.4
walked	79	77.2	9.2	8.4

Table 2: The number of calling each part of the unification routines. *Naive* shows the number of node unification operations in the naive unification algorithm (corresponds to `NODE_UNIFY` of my algorithm). `NODE_UNIFY` and `SEGMENT_UNIFY` are specified in Figure 6.

Test data	*Naive*	`NODE_UNIFY`	`SEGMENT_UNIFY`
credited	30929	256	5095
walked	65709	265	10603

equivalent parts. I implemented this method in LiLFeS, and achieved a speed-up of the unification process by a factor of 6.4 to 8.4. For realizing efficient NLP systems, I am currently building an efficient parser by integrating the packing method with the compilation method for HPSG (Torisawa and Tsujii, 1996). While the compilation method reduces the number of unification operations during parsing, it cannot prevent inefficiency caused by ambiguity. The packing method will overcome this problem, and will hopefully enable us to realize practical and efficient NLP systems.

References

Philippe Blache. 1997. Disambiguating with controlled disjunctions. In *Proc. International Workshop on Parsing Technologies.*

Philippe Blache. 1998. Parsing ambiguous structures using controlled disjunctions and unary quasi-trees. In *Proc. COLING-ACL'98*, pages 124–130.

Bob Carpenter. 1992. *The Logic of Typed Feature Structures.* Cambridge University Press.

Jochen Dörre and Andreas Eisele. 1990. Feature logic with disjunctive unification. In *Proc. 13th COLING*, volume 2, pages 100–105.

John Griffith. 1995. Optimizing feature structure unification with dependent disjunctions. In *Proc. Workshop on Grammar Formalism for NLP at ESSLLI-94*, pages 37–59.

John Griffith. 1996. Modularizing contexted constraints. In *Proc. COLING'96*, pages 448–453.

Kôiti Hasida. 1986. Conditioned unification for natural language processing. In *Proc. 11th COLING*, pages 85–87.

Robert T. Kasper and William C. Rounds. 1986. A logical semantics for feature structures. In *Proc. 24th ACL*, pages 257–266.

Takaki Makino, Minoru Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. 1998. LiLFeS — towards a practical HPSG parser. In *Proc. COLING-ACL '98*, pages 807–811.

Yusuke Miyao, Kentaro Torisawa, Yuka Tateisi, and Jun'ichi Tsujii. 1998. Packing of feature structures for optimizing the HPSG-style grammar translated from TAG. In *Proc. TAG+4 Workshop*, pages 104–107.

Mikio Nakano. 1991. Constraint projection: An efficient treatment of disjunctive feature descriptions. In *Proc. 29th ACL*, pages 307–314.

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

Yuka Tateisi, Kentaro Torisawa, Yusuke Miyao, and Jun'ichi Tsujii. 1998. Translating the XTAG English grammar to HPSG. In *Proc. TAG+4 Workshop*, pages 172–175.

Kentaro Torisawa and Jun'ichi Tsujii. 1996. Computing phrasal-signs in HPSG prior to parsing. In *Proc. 16th COLING*, pages 949–955.