Feature Structure Based Semantic Head Driven Generation

Gen-ichiro KIKUI
ATR Interpreting Telephony Research Laboratories
2-2 Hikari-dai, Seika-cho, Soraku-gun, Kyoto 619-02 JAPAN
kikui@atr-la.atr.co.jp

Abstract

This paper proposes a generation method for feature-structure-based unification grammars. As compared with fixed arity term notation, feature structure notation is more flexible for representing knowledge needed to generate idiomatic structures as well as general constructions. The method enables feature structure retrieval via multiple indices. The indexing mechanism, when used with a semantic head driven generation algorithm, attains efficient generation even when a large amount of generation knowledge must be considered. Our method can produce all possible structures in parallel, using structure sharing among ambiguous substructures.

1 Introduction

Practical generation systems must have linguistic knowledge of both specific expressions like idioms and general grammatical constructions, and they should efficiently produce surface strings applying that knowledge [1][2].

In order to satisfy the first requirement, our system employs a set of trees annotated with feature structures to represent generation knowledge. Each tree represents a fragment of a syntactic structure, and is paired with a semantic feature structure. We can describe idiomatic constructions, by making a tree which contains lexical specifications and is paired with a specific rather than general semantic structure. Because feature structures allow partial specification, we can encode generation knowledge ranging over multiple levels of generality in a uniform way.

However, notice that this property will be restricted if we use DCG or (fixed arity) term notation [1]. Suppose there is a generation knowledge structure whose syntactic part is "go on foot". The feature structure notation of its semantic part will be something like:

```
[[Reln GO]
 [Agent ?agent[]]
 [Instrument FOOT]].   (1)
```

while the term notation is :

instrument(go(Agent), foot) (2)

These two notations seem to be equivalent, but there is a crucial difference. A generation knowledge structure containing the feature-based semantics will still be unifiable *even if the semantic input to be unified contains additional material.* Thus the knowledge structure will be discovered and its syntactic information can be used for generation. By contrast, a term-based input with additional elements would not unify with the term-based semantic structure shown above. It would thus be necessary to create additional generation structures containing distinct (though partly overlapping) term-based semantic structures. Such additional structures are redundant and cause superfluous output.

For example, consider the augmented feature structure (3).

```
[[Reln GO]
 [Agent Ken]
 [Instrument FOOT]
 [Time   10:00am]]   (3)
```

It will indeed unify with (1) above. But term-based input semantic structure (4) will not unify with term-based semantic structure (2).

```
instrument(time(go(ken), 10:00am), foot).
                                    (4)
```

To unify (2), semantic structure (5) would also be required.

```
time(instrument(go(ken), foot), 10:00am).
                                    (5)
```

[1] The flexibility of structure notation compared to term notation is also discussed in [4].

For this reason, our generation knowledge consists of trees represented as feature structures. A tree can be substituted for a leaf node of another tree to form a larger structure. Thus, the tree can be regarded as a rule in a context-free feature-structure-based unification grammar.

The second requirement for a generation system is efficient creation of syntactic structures. This is the main topic of this paper. Our system is based upon Semantic Head Driven Generation [6], which is an efficient algorithm for unification based formalisms. However, this algorithm requires some additional mechanisms to efficiently retrieve relevant generation knowledge, because feature structures can not be easily indexed.

The algorithm presented here uses a multiple index network of feature structures to efficiently choose relevant generation knowledge from the knowledge base. The algorithm also uses an hypothetical node so as to efficiently maintain ambiguous structures during generation.

2 Phrase Description(PD)

Generation knowledge is represented as a set of trees annotated with feature structures. Each tree is called a Phrase Description (PD).

An example of a PD is shown in Figure.1.

```
Structure:
        (S AUX (NP PRON) VP)
Annotation:
 (S [[syn [[cat S][inv +]]]
    [sem [[reln REQUEST]
          [agen *SP*]
          [recp *HR*]
          [obje ?ACTION]]]])
 (AUX [[syn [[cat AUX][lex "would"]
       [v-morph PAST]]]])
 (NP  [[syn [[cat NP][case NOM]]]])
 (PRON  [[syn [[cat PRON][case NOM]
          [lex "you"]]]])
 (VP [[syn [[cat VP][v-morph BSE]]]
       [sem ?ACTION]])
```

Figure 1: an example of a PD

A PD consists of two parts: a structure definition and feature structure annotation (*Structure* and *Annotation* in Figure 1).

The structure definition defines the structure of a tree by using a list in which the first element corresponds to the mother node and the rest of the elements correspond to daughters. Each daughter may be a tree rather than a simple node.

The annotation part specifies the feature structure of each symbol appearing in the structure definition. A feature structure description can contain tags or variables (symbols with "?" as a prefix in the figure). The scope of a tag in a PD is the entire PD.

Each node should have a semantic and syntactic feature structure. The semantic feature on the root node of a PD represents the semantics of the PD; thus we call it the *semantic structure of the PD.*

Although the description represents a tree, it is the same as for a (partial) derivation structure of a unification-based CFG, because the current system does not allow *adjoining operations.* If the structure definition of every PD is restricted to mother-daughter relations only, the PD set is strictly equivalent to a unification-based CFG.

3 Generation Algorithm

Our algorithm is an efficient extension of Semantic Head Driven Generation. The major extensions are: 1) it handles feature structures directly, and 2) it creates all possible phrase structures in parallel. These extensions are embodied mainly in the PD activation and ambiguity handling mechanisms discussed in this section.

3.1 Overview of the algorithm

The main part of the generation process is expansion process, which iterates through *expanding node selection*, *activation*, *precombination*, and *application*, using an *expanding node agenda.*

Input to the process is a feature structure containing syntactic, semantic and pragmatic features as an initial constraint on the root node.

The expanding node agenda contains the unlexicalized leaf nodes of the tree under creation. At the beginning of the process, it contains only one node, which has the feature structure given as an initial constraint.

The expanding node selection step picks up one node, say *expanding node*, from the agenda. If no node is picked up, the expansion process stops.

The PD activation step activates all PD's whose semantic structures *subsume* the semantic structure of the expanding node.

The precombination step makes PD sequences from activated PD's to satisfy some constraints.

The application step instantiates the PD sequence(s) and applies it to the expanding node.

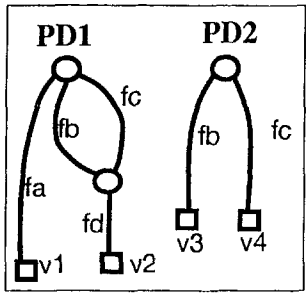

Figure 2: an example of dags

It also pushes unlexicalized leaf nodes into the expanding node agenda.

3.2 Expanding Node Selection

The expanding node selection step is for fetching one node from the expanding node agenda. From among the nodes whose semantic feature has been instantiated, one is chosen. In this step, if the fetched node satisfies some termination conditions (if, for instance, it satisfies the conditions for slash termination), the node is discarded (i.e., not expanded any more). If the agenda is empty or contains no node with an instantiated semantic feature, the expansion process stops.

3.3 Activation

This step is responsible for selecting all PD's whose semantic structures subsume the semantic structure of an expanding node. The selection is done by traversing a multiple index network of PD's called the *PD net*.

3.3.1 Compiling PD's

A set of PD's are pre-compiled into a PD net. Suppose there are two PD's whose semantic structures[2] are defined as the dags (i.e. *directed acyclic graphs*) in Figure 2. In the figure, fa,fb,fc,... and v1,v2,.. represent arc labels (feature names) and atomic values respectively. These PD's are compiled to the PD net shown in Figure 3.

The net has two kinds of nodes: *path nodes*(p_i), and *PD nodes* (PD_j). These nodes are linked by three kinds of labeled directed arcs:

[2] *The semantic feature of a PD.* is a semantic feature on the root node of the PD

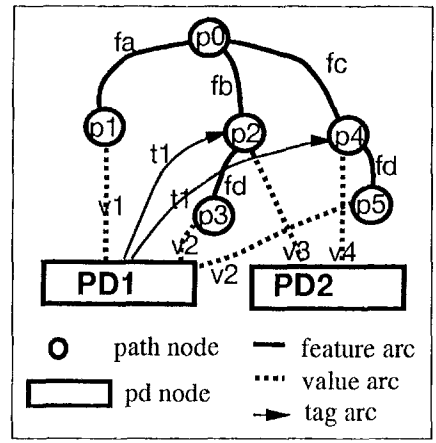

Figure 3: an example of PD net

feature arcs(bold lines), value arcs(dashed), and tag arcs(with arrows).

A path node is used to represent a particular feature path in one or more feature structure.

As shown in Figure 3, path nodes are linked by bold feature arcs to form a tree. The sequence of arc labels from the root path node to a path node p_i is the *path* of p_i. In Figure 3, p3 and p5 show paths (fb fd) and (fc fd) respectively.

Each PD node (rectangle) corresponds to a particular PD, which may have value arcs and tag arcs.

- Value Arcs: Which PD's contain which atomic values along certain paths ?

A PD node may be linked to path nodes with value arcs.

If a (rectangular) PD node is linked to a (round) path node pn with a dashed value arc labeled v, then following the path leading to pn yields atomic value v in that PD.

Consider the dashed value arc v1 in Figure 3. It indicates that following path fa in PD_1 yields an atomic value v1. This is just the situation depicted in Figure 2.

- Tag Arcs: In a given PD, which paths share a certain feature structure as a value ?

A PD node may also be linked to path nodes with tag arcs.

If two tag arcs have the same label and they connect a PD node to two path nodes, say pn1 and pn2, then the feature structure of that PD has a substructure which is the value of both paths, that of pn1 and pn2.

For example, the two tag arcs from rectangular PD1 node labeled "t1" in Figure 3 show that the semantic structure of PD1 has a substructure serving as the value of (fb) and (fc).

3.3.2 Traversing the PD net

The data structure of nodes and arcs are shown in Figure 4.

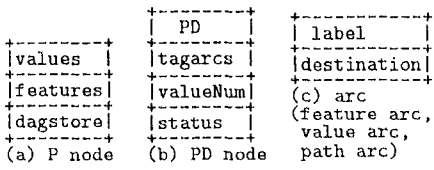

Figure 4: Node and Arc structures

A path node has three slots: *values*, *features*, and *dagstore*. The values slot and the arcs slot contain value arcs and feature arcs respectively. The dagstore slot is initially empty; later it holds a pointer to a dag which passed the path node.

Each PD node has a *PD slot*, a *tagarcs slot*, a *valueNum slot*, and a *status slot*. The PD slot contains a pointer to the represented PD itself. The tagarcs slot contains the data structure of the tagarcs (see below). The valueNum slot has the number of value arcs to the PD node. For example, the value of the number slot of PD1 node in Figure3 is 3, because the node has one value arc labeled v1 and two value arcs labeled v2. The status slot holds integer and is initially set to 0.

Every type of arc has two slots, 'label' and 'destination'. 'Label' is an atomic symbol which labels the arc, and 'destination' is a pointer to the node which is the destination of the arc.

We use the PD net as a dataflow net. The entry point of the net is the root path node and the token which flows is a dag of a semantic feature structure.

The action of a path node is shown in Figure 5. "failure" means there is no PD whose semantic structure subsumes the given dag. Thus the entire retrieval process fails.

The action of a PD node is shown in Figure 6. The status is incremented each time the node receives a token. As a result, if all atomic values in the semantic structure of the PD are satisfied, the status becomes equal to the valueNum (that is the number of atomic values). Once this is detected, then unifiability of shared structure is checked by calling the *tagtest* procedure.

Tagtest tests unifiability of the dags in the dagstores of p(ath) nodes connected by tag arcs with the same label. In Figure 3, if the status of PD1 becomes 3 and if the dag in p2 and the dag in p4 are identical, then the PD becomes *active*. That is, the PD has been found to subsume the generation input. It may or may not actually be applied, depending on later events.

```
PROCEDURE PDnodeAction(pdNode, value)
pdNode.status ← pdNode.status + 1 ;
IF pdNode.status = pdNode.valueNum
   and tagtest(pdNode.tagarcs) = T THEN
       activate(pdNode.PD) ;
```

Figure 6: Procedure of a PD node

If there is a PD node whose valueNums is 0 (i.e. No atomic value is specified in the semantic structure), node action of the PD node is invoked after dataflow is terminated.

```
PROCEDURE PathNodeAction(pathnode, dag)
pathnode.dagstore ← the pointer of dag ;
IF dag is atomic value type THEN
   validValueArcs ← {arc | arc ∈ pathnode.values, arc.label = dag.value} ;
   IF validValueArcs ≠ φ THEN
      FOR EACH arc IN validValueArcs DO
      PDnodeAction(arc.pdnode, dag) ;
   ELSE failure ;
ELSE IF dag is complex value type THEN
   FOR EACH arc IN dag.value DO
      IF Searcharc(arc.label, pathnode.featureArcs)
         ≠ Nil THEN
         PathNodeAction(Searcharc(arc.label, pathnode.featureArcs), arc.destination) ;
      ELSE failure
```

Figure 5: Procedure of a path node

3.4 Precombination

The precombination step is responsible for making sequences of PD's from activated PD's under certain constraints. A PD sequence is a rough plan for a potential structure which can realize a semantic specification of the node being expanded [3]. If no sequence is obtained, the ambiguity resolution process, discussed later, is invoked.

We divide PD's into two groups: *propagation type* and *non-propagation type*. A propagation type PD has has one *propagation node*. A propagation node of a PD is a leaf node whose semantic structure is identical with the semantic structure of the root node of the PD [4]. The rest of the PD's, which have no propagation nodes, are classified as non-propagation type PD's. This distinction is an extension of Shieber's chain rule and non-chain rule distinction.

A PD sequence $PD_1.....PD_n$ must satisfy the following constraints.

1. semantic structure sharing constraints

   (a) $PD_i (1 \leq i < n)$ is a propagation PD,

   (b) PD_n is a non-propagation PD,

   Under these constraints, the system can make a partial phrase structure by unifying the propagation node of PD_i with the root node of PD_{i+1}. The root node of the created structure contains the unified semantic structure of all semantic structures of PD's in the sequence.

2. local unifiability constraints

   (a) the root node of PD_1 is unifiable with the expanding node

   (b) PD_i and PD_{i+1} are connectable

   where PD_i is *connectable* to PD_j if PD_i is a propagation PD, and the propagation node of PD_i is unifiable with the root node of PD_j.

   These constraints are necessary conditions for unifiability throughout the entire PD sequence, which is called the *global unifiability of a PD sequence*. In contrast to such *global unifiability* constraints, the *local unifiability* can be pre-computed, or compiled, before generation process.

3. covering constraint

   Let fs1 be the unified semantic structure of all semantic structures of PD's in the sequence. Fs1 must contain every *top feature* of the semantic structure of the expanding node, where a *top feature* of a feature structure is defined as a top-level feature name.

   The covering constraint ensures *complete* generation [6]. If the constraint is not satisfied, a given semantic structure may not be completely realized in the generation result. For example, if an input semantic strucure is (3) (in Section 1) and the unified semantic structure of a PD sequence is (1), then the resulting PD sequence lacks the locative phrase for the "time" feature, which will not appear in the generation result.

4. disjointness constraints

   For each PD (PD_i), there is no other PD $(PD_j (i \neq j))$, such that PD_i has a top arc whose label is included in the set of top arcs of PD_j. The definition of top arc is given above.

   If this constraint is not satisfied, the generation result may contain duplicated or invalid expressions. For example, if a PD sequence contains two distinct PD's each of which is for a locative adjunct and has a "time" feature on the top level, the generation result will have two locative adjuncts for one semantic feature (i.e. the "time" feature).

   The disjointness constraint also ensures *compact* generation. Suppose a coherent and complete generator produces a string w, and the grammar assigns a semantic structure *fso* to w using a set of rules R. String w is *minimal* if every sub-structure of *fso* is supplied from one rule in R. The generator is *compact* if any string w is minimal.

In general, completeness, and compactness cannot actually be judged until the entire generation finishes. Thus the last two constraints (3 and 4) do not really guarantee completeness and compactness; rather, they help to limit search space in a practical way.

3.5 PD Application

The PD application step is responsible for creating phrase structure(s) from PD sequence(s) and attaching them to the expanding node. In this section, we restrict ourselves to the simple case

[3] A PD sequence is roughly corresponds to a *bottom-up chain* in [6]

[4] Our current system does not allow PD's with multiple semantic head

such that there is only one PD sequence obtained during the previous step. The case of multiple PD sequences, (i.e., *generation ambiguity*), will be discussed in the next section.

First, the module connects all PD's in the PD sequence $PD_i ... PD_n$ by unifying the propagation node of PD_i with the root node of PD_{i+1}. All unification operations are quasi-destructive, or temporal [7]. The result of the unification is valid until the module copies it (see below).

If this entire unification process succeeds (i.e., if every PD in the sequence can indeed be unified, and the sequence thus proves to be *globally unifiable*; see 3.7), then the module makes a copy of the unified PD sequence. Otherwise *expansion failure* (see next section) is signified. The copy, which is a phrase structure tree, is called an *instance of the PD sequence.*

Then the module attaches (unifies) the instantiated PD sequence to the expanding node.

Finally, the system stores in the expanding node agenda leaf nodes of expanded structures which have no lexical feature values. .

3.6 Ambiguity Handling

3.6.1 Ambiguity packing

If multiple PD sequences are applicable to an expanding node, the substructure of the expanding node can not be uniquely determined, because each PD sequence indicates only an hypothesis for the potential substructure.

The system maintain these hypotheses in a special *hypotheses slot* on the undetermined expanding node.

For each PD sequence, a copy of the expanding node called an *hypothesis node* is created. These copies are stored into the *hypotheses slot* of the original expanding node. Then the system applies each PD sequence to the corresponding hypothesis node, as described in the previous section, and continues expansion.

In Figure 7, three subtrees in the "hypo" slot on the undetermined node have been created for the hypothetical PD sequences.

The hypothetical PD sequences are not unified with the original expanding node, but unified with copies of the expanding node. This prevents the original feature structure of the undetermined node from being modified by further expansion of the hypothetical structures (T1-T3 in Figure 7).

The further expansion sometimes makes an hypothesis node inconsistent with the original

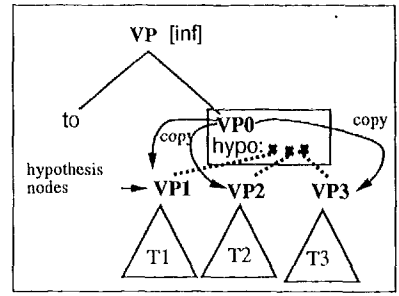

Figure 7: generation ambiguity

node. This is detected in the ambiguity resolution process described in the next section.

3.6.2 Expansion Failure and Ambiguity Resolution

Expansion failure occurs when:

1. no PD is activated in the PD activation, or
2. no PD sequences are obtained in the precombination, or
3. no PD sequences satisfy global connectability in the application.

The failure signifies that the feature structure of the current expanding node is inconsistent with a set of PD's given as generation knowledge.

The module searches for the nearest (i.e., lowest) hypothesis node (N_h) dominating the failed expanding node and deletes N_h from the hypotheses slot containing it.

If the number of hypothetical structures in the hypotheses slot of an undetermined node (N_u) becomes one, then N_u and the root node of the remaining structure in the hypotheses slot are unified. If the unification fails, ambiguity resolution continues recursively upward.

An example of ambiguity resolution is illustrated in Figure 8. The values of the hypotheses slot of node VP are the hypothetical nodes VP1, VP2, and VP3, corresponding to hypothetical trees T1, T2, T3 respectively. If expansion failure occur in T1 and T2, VP1 and VP2 are removed from the hypothesis slot. Then, VP3 is unified with VP, because there is only one hypothesis node left in the slot VP node.

If there is no hypothesis node dominating the failed expansion node, the entire generation process fails.

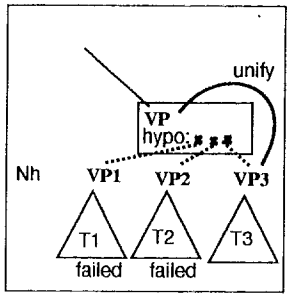

Figure 8: an illustration of generation ambiguity resolution

3.7 Postprocess

Expansion halts when no node is selected in the expanding node selection step. This does not necessarily mean the agenda is empty, because there may be some nodes without instantiated semantic structure.

How do such semantically empty nodes arise? The problem is that feature structures within hypothetical nodes are not allowed to unify with the feature structure on the "real" dominating node.

The solution is: for each hypothetical node, we create a complete tree using copies of the "real" dominating structure. Feature structures can then be permitted to unify with dominating structures. Then, the system collects all unlexicalized leaf nodes as initial values of the expanding node agenda and starts the normal expansion loop again.

4 Concluding Remarks

A semantic head driven generation method based on feature structures is proposed in this paper. This method efficiently generates all possible phrase structures from a given semantic feature structure. The method involves multiple indexing of feature structures and a precombination mechanism. These mechanisms constrain applicable grammatical knowledge before instantiation; thus the method eliminates the copying of feature structures, which consumes computing resources.

The proposed grammar notation is appropriate for describing idiomatic phrase structures easily. To make the best use of the notation, we are extending the algorithm so that it can perform adjunct operation [9].

The algorithm is implemented in SL-Trans, a spoken language translation system [8].

Acknowledgments

The author would like to thank Mark Seligman for helpful comments on this paper and also would like to thank Akira Kurematsu, Tsuyoshi Morimoto and other members of ATR for their constant help and fruitful discussions.

References

[1] Hovy, E.H., "Generating Natural Language Under Pragmatic Constraints",Ph.D.Dissertation, Yale Univ., 1987

[2] Jacobs, P.S., "A generator for natural language interfaces", In D.D.McDonald et al., editors, Natural Language Generation Systems, Chapter 7, Springer-Verlag, 1988

[3] Shieber,S. M., "An Introduction to Unification-Based Approaches to Grammar", CSLI, 1986

[4] Knight, K., "Unification: A Multidisciplinary Survey", ACM Computing Surveys, Vol.21, No.1, 1989

[5] Pollard,C. et al., "Information-based Syntax and Semantics Volume 1 Fundamentals", CSLI, 1987

[6] Shieber, S.M. et al., "A Semantic-Head-Driven Generation Algorithm", In Proceedings of 27th ACL, 1989

[7] Tomabechi, H., "Quasi-Destructive Graph Unification", In Proceedings of 29th ACL, 1991.

[8] Morimoto, T., et al. "A Spoken Language Translation System : SL-TRANS2". In Proceedings of COLING'92, 1992.

[9] Vijay-Shanker, K. et al., "Feature Structure Based Tree Adjoining Grammars", in Proceedings of COLING'88, 1988