ABSTRACT
The revised graph unification algorithms presented here are more efficient because they reduce the amount of copying that was necessary because of the assumption that data-structure sharing in inputs occurs only when feature-structure sharing occurs.

1 INTRODUCTION
Constraint-based linguistic frameworks use logical systems called feature logics (Kasper & Rounds, 1986; Shieber, 1980; Smolka, 1988), which describe linguistic objects by using logical formulas called feature descriptions that have as their models feature structures or typed feature structures. Shieber (1989) argued that if the canonical models of finite formulas of a feature logic were themselves finite, we could use them to compute over instead of theorem-proving over the formulas themselves. This would be advantageous if we had efficient algorithms for manipulating the canonical models.

The most important operation on models—feature structures or typed feature structures—is combining the information two models contain. This operation is traditionally called unification, although recently it has come to be more suitably called informational union. This unification operation is significant not only theoretically but also practically because the efficiency of systems based on constraint-based formalisms depends on the (typed) feature structure unification and/or feature description unification algorithms they use. This dependency is especially crucial for monostylistic formalisms—that is, formalisms which use only (typed) feature structures such as HPSG (Pollard & Sag, 1987) and JPSG (Gunji, 1987).

The efficiency of (typed) feature structure unification has been improved by developing algorithms that take as their inputs two directed graphs representing (typed) feature structures, copy all or part of them, and give a directed graph representing the unification result. These algorithms are thus called graph unification. Previous research has identified graph copying as a significant overhead and has attempted to reduce this overhead by lazy copying and structure sharing.

Unification algorithms developed so far, however, including those allowing structure sharing seem to contradict structure sharing because they assume the two input graphs never share their parts with each other. This "structure sharing" assumption prevents the initial data structures from sharing structures for representing linguistic principles and lexical information even though many lexical items share common information and such initial data structure sharing could significantly reduce the amount of data structures required, thus making natural language systems much more efficient. Furthermore, even if the structure sharing assumption holds initially, unification algorithms allowing structure sharing can yield situations that violate the assumption. The ways in which such unification algorithms are used are therefore restricted and this restriction reduces their efficiency.

This paper proposes a solution to this "structure sharing problem" and provides three algorithms. Section 2 briefly explains typed feature structures, Section 3 defines the structure sharing problem, and Section 4 presents key ideas used in solving this problem and provides three graph unification algorithms that increase the efficiency of feature structure unification in constraint-based natural language processing.

2 TYPED FEATURE STRUCTURES
The concept of typed feature structures augments the concept of feature structures. A typed feature structure consists of a set of feature-value pairs in which each value is a typed feature structure. The set of type symbols is partially ordered by subsumption ordering $\leq_T$ and constitutes a lattice in which the greatest element $\top$ corresponds to "no information" and the least element $\bot$ corresponds to "over-defined" or "inconsistency." For any two type symbols $a, b$ in this lattice, their least upper bound and greatest lower bound are respectively denoted $a \lor_T b$ and $a \land_T b$.

Typed feature structures are represented in matrix notation as shown in Fig. 1, where $\mathbf{syn}$, $\mathbf{agr}$, $\mathbf{sg}$, and $\mathbf{3rd}$ are type symbols; $\mathbf{agree}$, $\mathbf{num}$, $\mathbf{per}$, and $\mathbf{subj}$ are feature symbols; and $X$ is a tag symbol. A feature-address—that is, a finite (possibly empty) string of feature symbols—is used to specify a feature value of an embedded structure. In Fig. 1, for example, the structure at the feature-address $\mathbf{agree} \cdot \mathbf{num}$, where $\cdot$ is the concatenation operator, is said to have $\mathbf{sg}$ as its type symbol. The root feature-address is de-
noted by 't.' To specify token-identity in matrix notation, a tag symbol is used: feature-address values with the same tag symbol are token-identical, and those feature-addresses with the token-identical value are said to corefer. In Fig. 1, the feature-addresses agree and subj · agree corefer.

A typed feature structure is also represented by a rooted, connected, directed graph within which each node corresponds to a typed feature structure and is labeled with a type symbol (and, optionally, a tag symbol) and each arc corresponds to a feature-value pair and is labeled with a feature symbol. Fig. 2 illustrates the graph representation of the typed feature structure whose matrix notation is shown in Fig. 1. In a graph representation, the values at corresponding feature-addresses— that is, token-identical values—are represented by the same node.

The set of typed feature structures is also partially ordered by a subsumption ordering that is an extension of the subsumption ordering on the set of type symbols. A typed feature structure \( t_1 \) is less than or equal to \( t_2 \) (written as \( t_1 \preceq t_2 \)) if and only if \( t_1 \) is inconsistent (that is, if it includes the type symbol \( \perp \)) or (i) \( t_1 \)'s type symbol \( a_1 \) is less than or equal to \( t_2 \)'s type symbol \( a_2 \) (\( a_1 \preceq t_2 a_2 \)); (ii) each feature \( f \) of \( t_2 \) exists in \( t_1 \) and has a value \( v_2 f \) such that its counterpart \( t_1 f \) is less than or equal to \( t_2 f \); and (iii) each coreference relation holding in \( t_2 \) also holds in \( t_1 \).

This subsumption ordering serves as the basis for defining two lattice operations: generalization (the least upper bound or join) and unification (the greatest lower bound or meet).

Typed feature structures have been formalized in several ways, such as by using \( \psi \)-types (Ait-Kaci, 1986).

### 3 THE STRUCTURE SHARING PROBLEM

#### 3.1 Graph Unification Algorithms

The destructive unification algorithm presented by Ait-Kaci is the starting point in increasing the efficiency of graph unification. It is a node-merging process that uses the Union-Find algorithm, which was originally developed for testing finite automata equivalence (Hopcroft & Karp, 1971), in a manner very similar to that of the unification algorithm for rational terms (Huet, 1976). Given two root nodes of graphs representing (typed) feature structures, this algorithm simultaneously traverses a pair of input nodes with the same feature-address, putting them into a new and larger coreference class, and then returns the merged graph.

Since the destructive unification process modifies its input graphs, they must first be copied if their contents are to be preserved. Nondeterminism in parsing, for example, requires the preservation of graph structures not only for initial graphs representing lexical entries and phrase structure rules but also for those representing well-formed intermediate structures. Although the overhead for this copying is significant, it is impossible to represent a resultant unified graph without creating any new structures. Unnecessary copying, though, must be identified and minimized. Wroblewski (1987) defined two kinds of unnecessary copying - over-copying (copying structures not needed to represent resultant graphs) and early-copying (copying structures even though unification fails) - but this account is flawed because the resultant graph is assumed to consist only of newly created structures even if parts of the inputs that are not changed during unification could be shared with the resultant graph. A more efficient unification algorithm would avoid this redundant copying (copying structures that can be shared by the input and resultant graphs) (Kogure, 1990). To distinguish structure sharing at the implementation level from that at the logical level (that is, coreference relations between feature-addresses), the former is called data-structure sharing and the latter is called feature-structure sharing (Tomabechi, 1992).

The key approaches to reducing the amount of structures copied are lazy copying and data-structure sharing. For lazy copying, Karttunen (1986) proposed a reversible unification that saves the original contents of the inputs into preallocated areas immediately before destructive modification, copies the resultant graph if necessary, and then restores the original contents by undoing all the changes made during unification. Wroblewski (1987), on the other hand, proposed a nondestructive unification with incremental copying. Given two graphs, Wroblewski's algorithm simultaneously traverses each pair of input nodes with the same feature-address and creates a common copy of the input nodes. The nondestructive unification
algorithm for typed feature structures uses the data structures shown in Fig. 3.

The algorithm connects an input node and its copy node with a copy link—that is, it sets the copy node as the input’s copy field value. The link is meaningful during only one unification process and thus enables nondestructive modification. Using an idea similar to Karttunen’s, Tomabechi (1991) proposed a quasi-destructive unification that uses node structures with fields for keeping update information that survives only during the unification process.

Unification algorithms allowing data-structure sharing (DSS unification algorithms) are based on two approaches: the Boyer and Moore approach, which was originally developed for term unification in theorem-proving (Boyer & Moore, 1972) and was adopted by Pereira (1986), and the lazy copying suggested by Karttunen and Kay (1986). Recent lazy copying unification algorithms are based on Wroblewski’s or Tomabechi’s schema: Godden (1990) proposed a unification algorithm that uses active data structures, Kogure (1990) proposed a lazy incremental copy graph (LING) unification that uses dependency-directed copying, and Emele (1991) proposed a non-destructive unification (LING) that uses chronological dereference. These algorithms are based on Wroblewski’s algorithm, and Tomabechi (1992) has proposed a data-structure-sharing version of his quasi-destructive unification.

3.2 The Structure Sharing Problem

The graph unification algorithms mentioned so far—perhaps all those developed so far—assume that data-structure sharing between two input structures occurs only when feature-structure sharing occurs between feature-addresses they represent. This “structure sharing” assumption prevents data-structure sharing between initial data structures for representing linguistic principles and lexical information even though many lexical items share common information. For example, many lexical items in a traditional syntactic categories such as noun, intransitive verb, transitive verb, and so on share most of their syntactic information and differ in their semantic aspects such as semantic sortal restriction. Such initial data-structure sharing could significantly reduce the amount of data structures required and could therefore reduce page-swapping and garbage-collection and make natural language processing systems much more efficient.

Furthermore, even if the structure sharing assumption holds initially, applying a DSS unification algorithm in natural language processing such as parsing and generation can give rise to situations that violate the assumption. Consider, for example, JPSG-based parsing. There are only a few phrase structure rules in this framework and the Complement-Head Construction rule of the form “M → C H” is applied very frequently. For instance, consider constructing a structure of the form [vP1, NP2 [vP1, NP1 V]]. When the rule is applied, the typed feature structure for the rule is unified with the structure resulting from embedding the typed feature structure for NP1 at the feature-address for the complement daughter in the rule (e.g., *dtr* · *cdtr*), and the unification result is then unified with the structure resulting from embedding the typed feature structure for V at the feature-address for the head daughter. Because not every substructure of the structure for the rule always changed during such a unification process, there may be some substructures shared by the structure for the rule and the structure for VP1. Thus, when constructing VP2 there may be unexpected and undesired data-structure sharing between the structures.

Let me illustrate what happens in such cases by using a simple example. Suppose that we use the non-destructive unification algorithm or one of its data-structure sharing versions, the LING or LIC algorithm. The nondestructive and LING unification algorithms use the data structures shown in Fig. 3, and the LIC algorithm uses the same data structures except that its node structure has no forward field. Consider unification of the typed feature structures t1 and t2 shown in Fig. 4(a). Suppose that t1 and t2 are respectively represented by the directed graphs in Fig. 4(b) whose root nodes are labeled by tag symbols X0 and X4. That is, t1’s substructure at feature-address f2 and t2’s substructure at f4 are represented by the same data structure while feature-structure sharing does not hold between them, and t1’s substructure at f3 and t2’s substructure at f2 are represented by the same data structure while feature-structure sharing does not hold between them. Each of the algorithms simultaneously traverses a pair of input nodes with the same feature-address both of the inputs have from the root feature-address to leaf feature-addresses, makes a common copy of them to represent the unification result of that feature-address, and connects the input and output nodes with copy links. For any feature-address that only one of the inputs has, the nondestructive unification algorithm copies the subgraph whose root is the node for that feature-address and adds the copied subgraph to the output structure, whereas the LING and LIC algorithms make the node shared by the input and output structures. In the case shown in Fig. 4(b) the root nodes of the inputs—nodes with the tag symbols X0 and X4—are first treated by creating a common copy of them (i.e., the output node with X0), connecting the input and output nodes with copy links, and setting b0 = a0 ∧ a4 as the copy’s isymbol value. Then the input nodes’ arc structures are treated. Suppose that the pair of f1 arcs is treated first. After the input nodes at feature-address f1 are treated in the same manner as the root nodes, the pair of f2 arcs is treated. In this case, t1’s node at f2 (labeled X2) already has a copy link because the node is also used as t2’s node at f2 so that the destination node of the link is used as this feature-address’s output node. Af-

---

3For the nondestructive unification algorithm, the node structure takes as its copy field value either NIL or a node structure only.

4In this algorithm each unification process has an integer as its process identifier and each node created in a process has the identifier as its generation field value. A copy link is meaningful only if its destination node has the current process identifier. Such a node is called ‘current.’

The technique used to control the lifetime of update data is the same as that of Wroblewski’s algorithm.
where, their destination nodes, $t_1$, finally, the LING and MC algorithms make the input and output structures share algorithm copies $\alpha_2$ and $\alpha_3$ arcs and adds them to the common label arcs are treated. The nondestructive unification algorithm copies $t_1$’s $f_3$ and $t_2$’s $f_3$ arcs and adds them to the output root node, whereas the LING and LIC algorithms make the input and output structures share their destination nodes. Finally, the LING and LIC algorithms obtain graph $g_3$, represented in matrix notation in Fig. 4(c) just over the correct result.

The nondestructive unification algorithm obtains the same typed feature structure. The reversible and the quasi-destructive unification algorithms are also unable to obtain the correct result for this example because these algorithms cannot represent two update nodes by using a single node. Thus, none of the efficient unification algorithms developed recently obtains the correct results for such a case. Avoiding such wrong unification results requires undesirable copying. We can, for example, avoid getting the wrong result by interleaving the application of any non-DSS unification algorithm between applications of a DSS unification algorithm, but such bypassing requires two unification programs and reduces the efficiency gain of DSS unification. This preclusion of useful data-structure sharing is referred to here as the “structure sharing” problem.

It has been shown that all the DSS unification algorithms mentioned above are subject to this problem even if the structure sharing assumption holds initially. Non-DSS unification algorithms are also subject to the problem because their inputs are created by applying not only the unification operation but also operations such as embedding and extraction, in most implementations of which data-structure sharing occurs between their input and output structures. Even non-DSS unification algorithms must therefore take such inputs into account, and this requires undesirable copying.

4 A SOLUTION TO THE STRUCTURE SHARING PROBLEM

4.1 Key Ideas

The example in Section 3 suggests that the structure sharing problem has two sources, which concern not only the incremental copying approach but also other approaches. The first source is the way of recording update information. In the incremental copying approach, this corresponds to the way of copying structures. That is, while calculating $t_1 \land t_2$ the incremental copying process does not distinguish between the copies created as the substructures of the left input $t_1$ and the copies created as the substructures of the right input $t_2$. As a result, a copy node of $t_1$’s node at feature-address $p$ can be used as a copy node of $t_2$’s node at a feature-address, and vice versa. In Fig. 4(b), for example, the copy of $t_2$’s node at $f_2$ is wrongly used as the copy of $t_1$’s node at $f_1$. This causes unexpected and wrong data-structure sharing in the resultant graph and this in turn causes unexpected and wrong feature-structure sharing in the resultant (typed) feature structure. In other approaches, such as the quasi-destructive approach, the source of the structure sharing problem is that each node structure has fields for keeping information on only two typed feature structures—one for the original and one for each of the two results.

One way to solve this problem is therefore to make each node keep information on three typed feature structures: in the incremental copying approach each node must have two copy fields, and in the quasi-destructive approach each node must have two sets of fields for updates.

The second source of the structure sharing problem is the method of data-structure sharing between input and output structures. Unexpected and wrong data-structure sharing may result if a node shared by the left and right inputs is used as part of the left input, intended to be shared between the left input and output, at the same time it is used as part of the right input, intended to be shared between the right input...
and output. In Fig. 4(b), for example, $t_1$’s node at feature-address $f_1$ is shared as $t_2$’s node at the same feature-address, and the same node as $t_3$’s node at $f_4$ is shared as $t_4$’s node at the same feature-address.

This problem can be solved easily by keeping information on data-structure sharing status; that is, by adding to the node structure a new field for this purpose and using it thus: when a unification algorithm makes a node shared (for example, between the left input and output), it records this information on the node; later when the algorithm attempts to make the node shared, it does this only if this data-structure sharing is between the left input and output.

### 4.2 Algorithms

This section first describes a non-DSS unification algorithm that discards the structure sharing assumption and thus permits initial data-structure sharing, and then it describes two DSS unification algorithms.

#### Revised Nondestructive Unification

This algorithm uses, instead of the node structure shown in Fig. 3, the node structure in Fig. 5. That is, the algorithm uses two kinds of copy links: copy for the left input and copy for the right input.

The revised nondestructive unification procedure for typed feature structures is shown in Figs. 6 and 7. Given two root nodes of directed graphs, the top-level procedure $\text{Unify}$ assigns a new unification process identifier, generation, and invokes $\text{Unify}_Aux$. This procedure first dereferences both input nodes. This dereference process differs from the original one in that it follows up forward and copy links for the left input node and forward and copy links for the right input node. This revised dereference process eliminates the first source of the structure-sharing problem. Then $\text{Unify}_Aux$ calculates the meet of the type symbol. If the meet is $\bot$, which means inconsistency, it finishes by returning $\bot$. Otherwise $\text{Unify}_Aux$ obtains the output node and sets the meet as its $\text{tsymbol}$ value. The output node is created only when neither input node is current; otherwise the output node is a current input node. Then $\text{Unify}_Aux$ treats arcs. This procedure assumes the existence of two procedures: $\text{Shared}_Arcs$ and $\text{Complement}_Arcs$. The former gives two lists of arcs each of which contains arcs whose labels exist in both input nodes with the same arc label order; the latter gives one list of arcs whose labels are unique to the first input node. For each arc pair obtained by $\text{Shared}_Arcs$, $\text{Unify}_Aux$ applies itself recursively to the value pair. And for each arc pair obtained by $\text{Complement}_Arcs$, it copies its value.

Let us compare the newly introduced cost and the effect of this revision. This revised version differs from the original in that it uses two dereference procedures that are the same as the original dereference procedure except that they use different fields. Thus, on the one hand, the overhead introduced to this revision is only the use of one additional field of the node structure. On the other hand, although this revised version does not introduce new data-structure sharing, it can safely treat data-structure sharing in ini-

---

**Fig. 5:** The node structure for the revised nondestructive unification.

**Fig. 6:** The revised nondestructive unification procedure (1).
PROCEDURE Get_Out_Node(node1, node2, tsymbol)
IF Current_p(node1) AND Current_p(node2)
THEN
node2.forward ← node1;
nodel.tsymbol ← tsymbol;
return(node1)
ELSE IF Current_p(node1)
THEN
node2.copy ← node2;
nodel.tsymbol ← tsymbol;
return(node1)
ELSE IF Current_p(node2)
THEN
node1.copy ← node2;
nodel.tsymbol ← tsymbol;
return(node2)
ELSE
newnode ← Create_Node();
nodel.copy ← newnode;
nodel.tsymbol ← tsymbol;
return(newnode)
ENDIF
ENDPROCEDURE

Fig. 7: The revised nondestructive unification procedure (2).

Revised LING Unification
LING unification is based on nondestructive unification and uses copy-dependency information to implement data-structure sharing. For a unique label arc, instead of its value being copied, the value itself is used as the output value and copy-dependency relations are recorded to provide for later modification of shared structures. This algorithm uses a revised Copy_Node procedure that takes as its input two node structures (node1 and node2) and one arc structure, arc1 where node1 is the node to be copied. The structure arc1 is an arc to node1, and node1 is an ancestor node of node1—that is, the node from which arc1 departs—and the revised procedure is as follows: (i) if node1'(the dereference result of node1) is current, then Copy_Node returns node1' to indicate that the ancestor node2 must be copied immediately; otherwise, (ii) Copy_Arcs returns node1' and if it returns several arc copies, Copy_Node creates a new copy node and then adds to the new node the arc copies and arcs of node1' that are not copied, and returns the new node to indicate the ancestor node2 having to be copied immediately; otherwise, (iii) Copy_Node registers the copy-dependency between the node1' and the ancestor node node2—that is, it adds the pair consisting of the ancestor node node2 and the arc arc1 into the copy field of node1'—and returns NIL to indicate that the ancestor must not be copied immediately. When a new copy of a node is needed later, this algorithm will copy structures by using the copy-dependency information in its copy field (in the revised Get_Out_Node procedure for the LING unification). It substitutes arcs with newly copied nodes for existing arcs. Thus the antecedent nodes are also copied.

The revised LING unification is based on the revised nondestructive unification and uses a node-structure consisting of the fields in the node structure shown in Fig. 5 and a new field reuse for indicating node and arc pairs—deps field (see Fig. 3). The technique used to control the lifetime of copy-dependency information is the same as that of copy information. That is, the deps field value is meaningful only when the generation value is equal to the unification process identifier.

PROCEDURE Copy_Arcs_L(node)
newarcs ← 0;
FOR arc IN node.arcs DO
newnode ← Create_Arc(arc.label, newarcs);
node.copy.value ← newnode;
newarcs +← newarcs
ENDFOR;
node.arcs.reuse ← reused;
return(NIL)
ELSE
newarcs ← Add_Arc(newnode, arc.label, newarcs.value)
node.arcs.reuse ← reused;
return(newarcs)
ENDIF
ENDPROCEDURE

PROCEDURE Copy_Node_L(node, arc, ancestor)
node.reuse ← reused;
node.copy.dcop笑意s ← gcneration;
node.copy.value ← Create_Copydep();
copydep.copyvalue ← generation;
copydep.deps ← \{ancestor, arc\};
node.copy.value ← copydep;
node.arcs.reuse ← reused;
return(NIL)
ENDIF
ENDPROCEDURE

Fig. 8: The new revised Copy_Node procedure.
ing data-structure sharing status. When the top-level unification procedure is invoked, it sets two new symbols to the two variables \textit{bused} and \textit{rused}. That a node structure has as its reuse field value the \textit{bused} value means that it is used as part of the left input, and that it has as its reuse value the \textit{rused} value means that it is used as part of the right input. The revised LING unification uses two new revised \textit{CopyNode} procedures, \textit{CopyNode.L} (shown in Fig. 8) and the analogous procedure \textit{CopyNode.R}. These procedures are respectively used to treat the left and right inputs and they differ from the corresponding original procedure in two places. First, instead of step (i) above, if \textit{node1}' (the dereference result of \textit{node1}) is current, \textit{CopyNode.L} (or \textit{CopyNode.R}) returns \textit{node1} to indicate that the ancestor, \textit{node2}, must be copied immediately. But if \textit{node1} has as its reuse field value the \textit{rused} (or \textit{bused}) value, it creates a copy of the whole subgraph whose root is \textit{node1}, and returns the copied structure also to indicate that the ancestor node must be copied immediately. Second, in step (iii), they register data-structure sharing status—that is, they set the \textit{bused} (or \textit{rused}) value to the reuse field of \textit{node1}, as well as register copy-dependency information. This revised LING unification ensures safety in data-structure sharing.

Again let us compare the newly introduced computational costs and the effect of this revision. The newly introduced costs are the additional cost of the revised dereference procedures (which is the same as in the previous one) and the cost of checking reuse status. The former cost is small, as shown in the discussion of the previous algorithm, and the latter cost is also small. These costs are thus not significant relative to the efficiency gain obtained by this revision.

Revised Quasi-Destructive Unification

The structure-sharing version of quasi-destructive unification keeps update information in the field meaningful only during the unification. After a successful unification is obtained, this algorithm copies the unification result and attempts data-structure sharing. This algorithm can be revised to ensure safety in data-structure sharing by using a \textit{node} structure including two sets of fields for update information and one reuse field and by checking node reuse status while copying.

5 CONCLUSION

The graph unification algorithms described in this paper increase the efficiency of feature structure unification by discarding the assumption that data-structure sharing between two input structures occurs only when the feature-structure sharing occurs between the feature-addresses they represent. All graph unification algorithms proposed so far make this assumption and are therefore required to copy all or part of their input structures when there is a possibility of violating it. This copying reduces their efficiency. This paper analyzed this problem and points out key ideas for solving it. Revised procedures for nondestructive unification, LING unification, and quasi-destructive unification have been developed. These algorithms make the use of feature structures in constraint-based natural language processing much more efficient. The key ideas in this paper can also be used to make the incremental graph generalization algorithm (Kogure, 1993) more efficient.

ACKNOWLEDGMENTS

I thank Akira Shimazu, Mikio Nakano, and other colleagues in the Dialogue Understanding Group at the NTT Basic Research Laboratories for their encouragement and thought-provoking discussions.

REFERENCES

Alf-Kari, H. (1986). An Algebraic Semantics Approach to the Effective Resolution of Type Equations. J. of Theor. Comp. Sci., 45, 293-351.

Broy, R. S., & Moore, J. S. (1972). The Sharing of Structure in Theorem-Proving Programs. In Meltzer, B., & Michie, D. (Eds.), Machine Intelligence Vol. 7, chap. 6, pp. 101-116. Edinburgh University Press.

Emele, M. (1991). Unification with Lazy Non-Redundant Copying. In Proc. of the 29th ACL, pp. 325-330.

Golden, K. (1990). Lazy Unification. In Proc. of the 28th ACL, pp. 186-187.

Gung, T. (1987). Japanese Phrase Structure Grammar. Reidel.

Hopcroft, J. E., & Karp, R. M. (1971). An Algorithm for Testing the Equivalence of Finite Automata. Tech. Rep. TR-71-114, Dept. of Comp. Sc., Cornell University.

Huet, G. (1976). Résolution d'Equations dans des Langages d'Ondre 1, 2, ..., w. Ph. D. thesis, Université de Paris VII.

Karttunen, L. (1986). D-PATR-- A Development Environment for Unification-Based Grammars. Tech. Rep. CSLI-86-61, CSLI.

Karttunen, L., & Kay, M. (1985). Structure Sharing Representation with Binary Trees. In Proc. of the 23rd ACL, pp. 133-136.

Kasper, R. T., & Rounds, W. C. (1986). A Logical Semantics for Feature Structure. In Proc. of the 24th ACL.

Kogure, K. (1989). Parsing Japanese Spoken Sentences based on HPSG. In Proc. of the Int. Workshop on Parsing Technologies, pp. 132-141.

Kogure, K. (1990). Strategic Lazy Incremental Copy Graph Unification. In Proc. of the 15th COLING, Vol. 2, pp. 223-228.

Kogure, K. (1991). Typed Feature Structure Generalization by Incremental Graph Copying. In Tрост, H. (Ed.), Feature Formalisms and Linguistic Ambiguity, pp. 130-158. Ellis Horwood.

Pereira, F. C. N. (1985). Structure Sharing Representation for Unification-Based Formalisms. In Proc. of the 23rd ACL, pp. 137-144.

Pollard, C., & Sag, I. (1987). An Information-Based Syntax and Semantics—Volume 1: Fundamentals. CSLI Lecture Notes No. 13, CSLI.

Shieber, S. M. (1989). Constraint-Based Grammar Formalisms—Parsing and Type Inference for Natural and Computer Languages. Ph. D. thesis, Stanford University.

Smolka, G. (1988). A Feature Logic with Subsorts. LILOG '83, IBM Deutschland.

Tomabechi, H. (1991). Quasi-Destructive Graph Unification. In Proc. of the 29th ACL, pp. 315-322.

Tomabechi, H. (1992). Quasi-Destructive Graph Unification with Structure-Sharing. In Proc. of the 13th COLING, pp. 446-446.

Wroblewski, J. A. (1987). Nondestructive Graph Unification. In Proc. of the 6th AAAI, pp. 582-587.