STRUCTURE SHARING PROBLEM AND ITS SOLUTION IN GRAPH UNIFICATION

Kiyoshi KOGURE

NTT Basic Research Laboratories
3-1 Morinosato-Wakamiya, Atsugi-shi, Kanagawa, 243-01 Japan
kogure@atom.ntt.jp

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, 1989; 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.[1] This dependency is especially crucial for monostratal formalisms—that is, formalisms which use only (typed) feature structures such as HPSG (Pollard & Sag, 1987) and JPSG (Gunji, 1987).[2]

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.

$$\mathbf{syn}\left[\begin{array}{ll} \mathit{agree}: & \mathsf{X} : \mathbf{agr}\left[\begin{array}{ll} \mathit{num}: & \mathbf{sg} \\ \mathit{per}: & \mathbf{3rd} \end{array}\right] \\ \mathit{subj}: & \mathbf{syn}\left[\mathit{agree}: \mathsf{X}\ \right] \end{array}\right]$$

Fig. 1: Matrix notation for a typed feature structure.

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 $\leq_{\mathcal{T}}$ 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 $\mathbf{a} \vee_{\mathcal{T}} \mathbf{b}$ and $\mathbf{a} \wedge_{\mathcal{T}} \mathbf{b}$.

Typed feature structures are represented in matrix notation as shown in Fig. 1, where **syn**, **agr**, **sg**, and **3rd** are type symbols; *agree*, *num*, *per*, and *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 $\mathit{agree} \cdot \mathit{num}$, where '$\cdot$' is the concatenation operator, is said to have **sg** as its type symbol. The root feature-address is de-

[1] For example, the TASLINK natural language system uses 80% of the processing time for feature structure unification and other computations required by unification, i.e., feature structure pre-copying (Godden, 1990).

[2] For example, a spoken-style Japanese sentence analysis system based on HPSG (Kogure, 1989) uses 90%–98% of the processing time for feature structure unification.

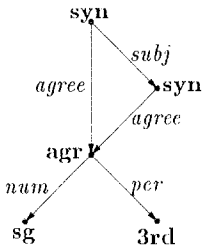


Fig. 2: Graph representation of a typed feature structure.

noted by 'c.' 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 coreferent 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 $t_1 \leq_t t_2$) if and only if t_1 is inconsistent (that is, if it includes the type symbol $\perp$) or (i) t_1's type symbol $\mathbf{a}_1$ is less than or equal to t_2's type symbol $\mathbf{a}_2$ ($\mathbf{a}_1 \leq_T \mathbf{a}_2$); (ii) each feature f of t_2 exists in t_1 and has a value $t_{2,f}$ 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 ψ-types (Aït-Kaci, 1986).

3 THE STRUCTURE SHARING PROBLEM

3.1 Graph Unification Algorithms

The destructive unification algorithm presented by Aït-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.

node structure	
tsymbol	⟨a type symbol⟩
arcs	⟨a set of *arc* structures⟩
generation	⟨an integer⟩
forward	*NIL* | ⟨a *node structure*⟩
copy	*NIL* | ⟨a *node* structure⟩ | ⟨a *copydep* structure⟩

arc structure	
label	⟨a feature symbol⟩
value	⟨a *node* structure⟩

copydep structure	
generation	⟨an integer⟩
deps	⟨a set of *node* and *arc* pairs⟩

Fig. 3: Data structures for nondestructive unification and LING unification.

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.[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.[4] 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.[5]

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 (1985); and the lazy copying suggested by Karttunen and Kay (1985). 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 lazy-incremental copying (LIC) unification 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 [$_{VP_2}$ NP_2 [$_{VP_1}$ NP_1 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 NP_1 at the feature-address for the complement daughter in the rule (e.g., *dtrs* · *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 VP_1. Thus, when constructing VP_2 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 nondestructive 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 t_1 and t_2 shown in Fig. 4(a). Suppose that t_1 and t_2 are respectively represented by the directed graphs in Fig. 4(b) whose root nodes are labeled by tag symbols X_0 and X_4. That is, t_1's substructure at feature-address f_2 and t_2's substructure at f_1 are represented by the same data structure while feature-structure sharing does not hold between them, and t_1's substructure at f_3 and t_2's substructure at f_4 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 X_0 and X_4—are first treated by creating a common copy of them (i.e., the output node with Y_0), connecting the input and output nodes with *copy* links, and setting $\mathbf{b}_0 = \mathbf{a}_0 \wedge_T \mathbf{a}_4$ as the copy's *tsymbol* value. Then the input nodes' *arc* structures are treated. Suppose that the pair of f_1 arcs is treated first. After the input nodes at feature-address f_1 are treated in the same manner as the root nodes, the pair of f_2 arcs is treated. In this case, t_1's node at f_2 (labeled X_2) already has a *copy* link because the node is also used as t_2's node at f_1 so that the destination node of the link is used as this feature-address's output node. Af-

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

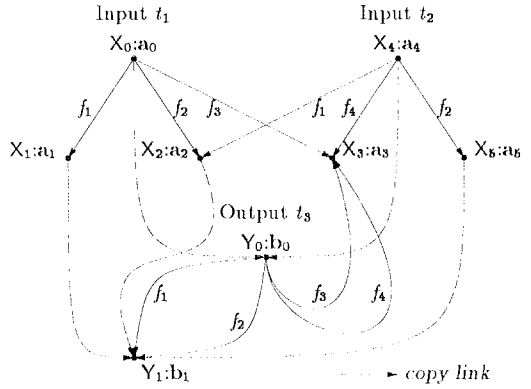
[4] In this algorithm each unification process has an integer as its process identifier and each node created in a process has the identifier as its *generation* field value. A *copy* link is meaningful only if its destination node has the current process identifier. Such a node is called 'current.'

[5] The technique used to control the lifetime of update data is the same as that of Wroblewski's algorithm.

$$t_1 \quad : \quad \mathrm{a}_0 \begin{bmatrix} f_1 : \mathrm{a}_1 \\ f_2 : \mathrm{a}_2 \\ f_3 : \mathrm{a}_3 \end{bmatrix},$$

$$t_2 \quad : \quad \mathrm{a}_4 \begin{bmatrix} f_1 : \mathrm{a}_2 \\ f_2 : \mathrm{a}_5 \\ f_4 : \mathrm{a}_3 \end{bmatrix}.$$

(a) Input typed feature structures.



(b) Snapshot of incremental graph unification allowing data-structure sharing.

$$t_3 \quad : \quad \mathrm{b}_0 \begin{bmatrix} f_1 : \mathrm{Y}_1 : \mathrm{b}_1 \\ f_2 : \mathrm{Y}_1 \\ f_3 : \mathrm{X}_3 : \mathrm{a}_3 \\ f_4 : \mathrm{X}_3 \end{bmatrix},$$

$$t_1 \wedge_t t_2 \quad : \quad \mathrm{b}_0 \begin{bmatrix} f_1 : \mathrm{b}_2 \\ f_2 : \mathrm{b}_3 \\ f_3 : \mathrm{a}_3 \\ f_4 : \mathrm{a}_3 \end{bmatrix}$$

where

$$\begin{aligned} \mathrm{b}_0 &= \mathrm{a}_0 \wedge_7 \mathrm{a}_4, \\ \mathrm{b}_1 &= \mathrm{a}_1 \wedge_7 \mathrm{a}_2 \wedge_7 \mathrm{a}_5, \\ \mathrm{b}_2 &= \mathrm{a}_1 \wedge_7 \mathrm{a}_2, \\ \mathrm{b}_3 &= \mathrm{a}_2 \wedge_7 \mathrm{a}_5. \end{aligned}$$

(c) Wrong graph unification output (t_3) and the correct unification of the inputs ($t_1 \wedge_t t_2$).

Fig. 4: An example of incorrect graph unification.

ter the common label arcs are treated, unique label arcs are treated. The nondestructive unification algorithm copies t_1's f_3 and t_2's f_4 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 t_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 $t_1 \wedge_t t_2$ 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 the result—whereas fields for keeping information on three typed feature structures are needed—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

node structure	
tsymbol	⟨a type symbol⟩
arcs	⟨a set of *arc* structures⟩
generation	⟨an integer⟩
forward	*NIL* | ⟨a *node* structure⟩
lcopy	*NIL* | ⟨a *node* structure⟩
rcopy	*NIL* | ⟨a *node* structure⟩

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

and output. In Fig. 4(b), for example, t_1's node at feature-address f_3 is shared as t_3's node at the same feature-address, and the same node as t_2's node at f_4 is shared as t_3'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: *lcopy* for the left input and *rcopy* 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 *Unify* assigns a new unification process identifier, *generation*, and invokes *Unify_Aux*. This procedure first dereferences both input nodes. This dereference process differs from the original one in that it follows up *forward* and *lcopy* links for the left input node and *forward* and *rcopy* links for the right input node. This revised dereference process eliminates the first source of the structure-sharing problem. Then *Unify_Aux* calculates the meet of the type symbol. If the meet is $\perp$, which means inconsistency, it finishes by returning $\perp$. Otherwise *Unify_Aux* obtains the output node and sets the meet as its *tsymbol* value. The output node is created only when neither input node is current; otherwise the output node is a current input node. Then *Unify_Aux* treats arcs. This procedure assumes the existence of two procedures: *Shared_Arc_Pairs* and *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 *Shared_Arc_Pairs*, *Unify_Aux* applies itself recursively to the value pair. And for each arc obtained by *Complement_Arcs*, it copies its *value*.

Let us compare the newly introduced cost and the

```
PROCEDURE Unify(node1, node2)
  generation ← generation + 1;
  return(Unify_Aux(node1, node2))
ENDPROCEDURE
PROCEDURE Unify_Aux(node1, node2)
  node1 ← Dereference_L(node1);
  node2 ← Dereference_R(node2);
  IF node1 = node2 AND Current_p(node1) THEN
    return(node1)
  ENDIF
  newtsymbol ← node1.tsymbol ∧_T node2.tsymbol;
  IF newtsymbol = ⊥ THEN
    return(⊥)
  ENDIF;
  newnode ← Get_Out_Node(node1, node2, newtsymbol);
  ⟨sarcs1, sarcs2⟩ ← Shared_Arc_Pairs(node1, node2);
  carcs1 ← Complement_Arcs(node1, node2);
  carcs2 ← Complement_Arcs(node2, node1);
  FOR ⟨sarc1, sarc2⟩ IN ⟨sarcs1, sarcs2⟩ DO
    newvalue ← Unify_Aux(sarc1.value, sarc2.value);
    IF newvalue = ⊥ THEN
      return(⊥)
    ELSE
      newvalue
        ← Add_Arc(newnode, sarc1.label, newvalue);
      IF newvalue = ⊥ THEN
        return(⊥)
      ENDIF
    ENDIF
  ENDFOR;
  IF newnode ≠ node1 THEN
    FOR carc IN carcs1 DO
      newvalue ← Copy_Node_L(carc.value);
      newnode
        ← Add_Arc(newnode, carc.label, newvalue)
    ENDFOR
  ELSE IF newnode ≠ node2 THEN
    FOR carc IN carcs2 DO
      newvalue ← Copy_Node_R(carc.value);
      newnode
        ← Add_Arc(newnode, carc.label, newvalue)
    ENDFOR
  ENDIF;
  return(newnode)
ENDPROCEDURE
PROCEDURE Dereference_L(node)
  IF Node_p(node.forward) THEN
    return(Dereference_L(node.forward))
  ELSE IF Curret_Node_p(node.lcopy) THEN
    return(Dereference_L(node.lcopy))
  ELSE
    return(node)
  ENDIF
ENDPROCEDURE
```

Fig. 6: The revised nondestructive unification procedure (1).

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-

```
PROCEDURE Get_Out_Node(node1, node2, tsymbol)
  IF Current_p(node1) AND Current_p(node2) THEN
    node2.forward ← node1;
    node1.tsymbol ← tsymbol;
    return(node1)
  ELSE IF Current_p(node1) THEN
    node2.rcopy ← node1;
    node1.tsymbol ← tsymbol;
    return(node1)
  ELSE IF Current_p(node2) THEN
    node1.lcopy ← node2;
    node2.tsymbol ← tsymbol;
    return(node2)
  ELSE
    newnode ← Create_Node();
    node1.lcopy ← newnode;
    node1.rcopy ← newnode;
    newnode.tsymbol ← tsymbol;
    return(newnode)
  ENDIF
ENDPROCEDURE
```

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

tial data structures. This can significantly reduce the amount of initial data structures required for linguistic descriptions, especially for lexical descriptions, and thus reduce garbage-collection and page-swapping.

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* is applied to *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 node having to be copied immediately; otherwise, (iii) *Copy_Node* registors 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.[6] 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.

```
PROCEDURE Copy_Node_L(node, arc, ancestor)
  node ← Derference_L(node);
  IF Current_p(node) THEN
    return(node);
  ELSE IF node.reuse = rused THEN
    return(Simple_Copy_Node_L(node))
  ENDIF
  newarcs ← Copy_Arcs_L(node);
  IF newarcs ≠ ∅ THEN
    newnode ← Create_Node();
    newnode.tsymbol ← node.tsymbol;
    node.lcopy ← newnode;
    FOR arc IN node.arcs DO
      newarc ← Find_Arc(arc.label, newarcs);
      IF Arc_p(newarc) THEN
        newvalue
          ← Add_Arc(newnode, arc.label, newarc.value)
      ELSE
        newvalue
          ← Add_Arc(newnode, arc.label, arc.value)
      ENDIF
    ENDFOR;
    return(newnode)
  ELSE IF Copydep_p(node.lcopy) AND
          node.lcopy.generation = generation THEN
    node.lcopy.deps
      ← node.lcopy.deps ∪ {⟨ancestor, arc⟩};
    node.reuse ← lused;
    return(NIL)
  ELSE
    copydep ← Create_Copydep();
    copydep.generation ← generation;
    copydep.deps ← {⟨ancestor, arc⟩};
    node.lcopy ← copydep;
    node.reuse ← lused;
    return(NIL)
  ENDIF
ENDPROCEDURE
PROCEDURE Copy_Arcs_L(node)
  newarcs ← ∅;
  FOR arc IN node.arcs DO
    newnode ← Copy_Node(arc.value, arc, node);
    IF Node_p(newnode) THEN
      newarc ← Create_Arc(arc.label, newnode);
      newarcs ← newarcs ∪ {newarc}
    ENDIF
  ENDFOR;
  return(newarcs)
ENDPROCEDURE
```

Fig. 8: The new revised *Copy_Node* procedure.

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 indicat-

[6] In the LING unificaiton algorithm, a *node* structure's *copy* field is used to keep either copy information or copy-dependency information. When the field keeps copy-dependency information, its value is a *copydep* structure consisting of an integer—*generation* field—and a set of *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.

ing data-structure sharing status. When the top-level unification procedure is invoked, it sets two new symbols to the two variables *lused* and *rused*. That a *node* structure has as its *reuse* field value the *lused* value means that it is used as part of the left input, and that it has as its *reuse* value the *rused* value means that it is used as part of the right input. The revised LING unification uses two new revised *Copy_Node* procedures, *Copy_Node_L* (shown in Fig. 8) and the analogous preocedure *Copy_Node_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 *node1'* (the dereference result of *node1*) is current, *Copy_Node_L* (or *Copy_Node_R*) returns *node1'* to indicate that the ancestor, *node2*, must be copied immediately. But if *node1'* has as its *reuse* field value the *rused* (or *lused*) value, it creates a copy of the whole subgraph whose root is *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 *lused* (or *rused*) value to the *reuse* field of *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 *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

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

Boyer, 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.

Godden, K. (1990). Lazy Unification. In *Proc. of the 28th ACL*, pp. 180–187.

Gunji, 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. Sci., Cornell University.

Huet, G. (1976). *Résolution d'Equations dans des Langages d'Ordre 1, 2, ..., ω*. 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 13th COLING, Vol. 2*, pp. 223–228.

Kogure, K. (1993). Typed Feature Structure Generalization by Incremental Graph Copying. In Trost, H. (Ed.), *Feature Formalisms and Linguistic Ambiguity*, pp. 139–158. 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 33, 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 14th COLING*, pp. 440–446.

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