TR-I-0129

戦略的遅延逐次複製グラフ単一化手法

Strategic Lazy Incremental Copy Graph Unification Method

小暮　潔

Kiyoshi Kogure

1990.1.26

Abstract

Two feature structure unification methods called the lazy incremental copy graph unification method and the strategic incremental copy graph unification method have been developed. The former method achives structure sharing with constant order data access time which reduces required memory. The latter method uses an early failure finding strategy which tries to unify substructures tending to first fail in unification. This method is based on stochastic data and reduces unnecessary computation. The two methods can be combined into a method called the strategic lazy incremental copy graph unification method. The combined method not only makes each feature structure unification efficient, but also reduces garbage collection and page swapping occurrences, thus increasing total efficiencies of TFS unification-based natural language processing systems such as a spoken Japanese sentence analysis system based on HPSG..

ATR 自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

Contents

1 Introduction

Various kinds of grammatical formalisms without transformation were proposed from the late 1970s through the 1980s [3,2,5,10,11]. These formalisms were developed independently and were made assure that they had common properties, that is, using feature structures and being based on their unification operation. These formalisms were applied in the field of natural language processing[6,1] and machine translation systems based on these formalisms were developed.

In such unification-based formalisms, feature structure (FS) unification, or unification of directed graphs (DG) representing FSs, is the most fundamental and significant operation. Efficiencies of systems based on such formalisms, such as natural language analysis and generation systems very much depend on their FS unification efficiencies. This dependency is especially crucial for lexicon-driven approaches such as HPSG[11] and JPSG[4] because large numbers of DG structures are used to represent rich lexical information and phrase structure information in terms of FSs. For example, a spoken Japanese sentence analysis module based on HPSG uses 90% $\sim$ 98% of the elapsed time in FS unification[1].

Several FS unification methods were proposed in [9,6,12]. Previous research identified DG copying as the most significant overhead. Wroblewski[12] proposed an incremental copy graph unification method to avoid overheads called '*over copying*'[2] and '*early copying*'[3]. He claimed that his method was at least as efficient as other unification methods.

However, the problem with his method is that a unification result graph consists only of newly created structures. This is not necessary because there are often input subgraphs which can be used as part of the result graph without any modification, or sharable parts between one of the input graphs and the result graph. Copying sharable parts is named '*redundant copying*'. A better method would minimize sharable part copying. The redundantly copied parts are relatively large when input graphs have few common feature paths. In natural language processing, such cases occur ubiquitously. For example, in combining a head and complement constituent, such cases occur quite frequently. Moreover, in Kasper's disjunctive feature description unification[7], such cases occur quite frequently in unifying definite and indefinite parts of disjuncts. Memory is wasted by such redundant copying and this causes frequent garbage collection

[1] Experiment results show this significance which the module becomes 5 $\sim$ 50 times faster by looking up a unification result table instead of applying FS unification.

[2] In destructive unification, copies are made of both input DGs. These copies are then ravaged by the unification method to build a result DG. This would appear to require new materials for two DGs in order to create just one new DG. A better method would be only to allocate enough memory for the resulting DG.

[3] In destructive unification, the input DGs are copied before unification is started. If the unification fails, some of the copying is a wasted effort. A better method would be to copy incrementally, so that if a failure occurred, only copying would only be minimal before the failure was detected.

and page swapping, which decrease total system effiency[4]. Developing a method which avoids memory wasting is very important.

Pereira's structure sharing FS unification method, which achieves structure sharing by using a data structure consisting of a skeleton part to represent original information, and an environment part to represent updated information, can avoid this problem. The skeleton part is shared by one of the input FSs and the result FS. Therefore, his method needs few new structures when two input FSs are different in size and which input FS is larger is known before unification.

However, his method can create skeleton–environment structures, for example, in recursively constructing large phrase structures from their parts, which are too deeply embedded. This embedding causes $O(\log d)$ graph node access overhead assembling the whole DG from the skeleton and updates in the environments where d is the number of nodes in the DG.

In this paper, an FS unification method is proposed which allows structure sharing with constant order node access time. This method achieves structure sharing by introducing lazy copying to Wroblewski's incremetal copy graph unification. The method is called the Lazy Incremetal Copy Graph unification method (the LING unification method in short).

The advantages of natural language processing systems based on declarative constraint rule descriptions in terms of FSs include:

1. rule writers are not required to describe control information such as constraint application order in a rule, and

2. rule descriptions can be used in different processing directions, i.e., natural language analysis and generation.

However, these advantages in describing rules are disadvantages in applying them because of the lack of control information. For example, when constructing a constituent from from its parts (e.g., a subject NP and a definite VP), unncessary computation can be reduced if the semantic component is assembled from its parts only after checking conditions such as grammatical agreements, which may fail. This is impossible in straightforward unification-based formalisms.

In contrast, in a procedure-based system which uses IF-THEN style rule, it is possible to construct the semantic structure (THEN part) after examining the agreements (IF part). Such a system has the advantage of processing efficiency but the disadvantage of lack of multi-directionality[5].

[4]For example, in the Spoken Japanese analysis module mentioned previously, analysis is much faster when neither garbage collection nor page swapping occur during analysis than when they do occur. Moreover, when a sentence is analyzed twice, the second analysis process takes much more elapsed time compared to the first process. Sometime, the second analysis process takes more than 5 times of that required by the first process.

[5]For example, in a topdown generation process, agreement features are not determined before lexical entries. To do so would be inefficient.

In this paper, some of the efficiency of the procedure-based system is introduced into an FS unification-based system. That is, an FS unification method is proposed which introduces a strategy called the Early Failure Finding Strategy (the EFF strategy) to make FS unification efficient. In this method, FS unification orders are not described by constraint writers (e.g., separating IF and THEN parts), but are controlled by learned tendencies of FS constraint satisfaction failures. This method is called the Strategic Incremental Copy Graph Unification (SING method).

These two unification methods can be combined and the combined method, called the Strategic Lazy Incremental Copy Graph unification method (SLING method), is used in a spoken Japanese sentence analysis module based on HPSG[8].

Section 2 explains a typed feature structure (TFS) unification method based on Wroblewski's method and then explains the problem with his method. The section also introduces the key idea of the EFF strategy which comes from observations of his method. Sections 3 and 4 introduce the LING method and the SING method, respectively.

2 Wroblewski's Incremental Copy Graph Unification Method and its Problems

In TFS unification based on Wroblewski's method, a DG is represented by using the **NODE** and **ARC** structures shown in Fig. 2. A **NODE** structure represents a TFS, and an **ARC** structure represents a feature–value pair. The **NODE** structure has the slots **TSYMBOL** to represent a type symbol, **ARCS** to represent a set of feature–value pairs, **GENERATION** to specify the unification process in which the structure has been created, **FORWARD** and **COPY**. That a **NODE** structure's **GENERATION** value is equal to the value of a global variable (e.g., ***GENERATION***) means that the structure has been created in the current unification process, or that the structure is '*current*'.

The characteristics which allow incremental copy are the **NODE** structure's two different slots **FORWARD** and **COPY** for representing forwarding relationships. A **FORWARD** slot value represents an eternal forwarding relationship while a **COPY** slot value represents a temporal relationship. When a **NODE** structure *node1* has a **NODE** structure *node2* as its **FORWARD** slot value, the other contents of the *node1* are *always* ignored and the contents of *node2* are used. If *node2* also has a **NODE** structure *node3* as its **FORWARD** value, the contents of the *node2* are ignoreed, too. However, when a **NODE** has a **NODE** structure as its **COPY** value, the contents of the **COPY** value are used only when the **COPY** value is current (and there is no **COPY** node). After the global variable is updated, all **COPY** slot values are ignored and both the newly created and original data can be accessed.

The unification procedure based on Wroblewski's method takes as its input two **NODE** structures which are roots of the DGs to be unified (See Fig. 2). The procedure copies **NODE** and **ARC** structures on the subgraphs of each input DG incrementally until a **NODE** structure with an empty **ARCS** slot value is found. This procedure can be illustrated with an example of the unification of two FSs as shown in Fig. 2 (a) and (b). The procedure first dereferences two input nodes (i.e., it follows up **FORWARD** and **COPY** slot values. See Fig. 2) and then calculates '*the most general specifier*' of their type symbol, or their '*meet*'[6], by retrieving the type symbol meet table. If the meet is $\perp$, which means inconsistency, the procedure finishes and returns $\perp$. Otherwise, the procedure obtains the output node with the meet as its **TSYMBOL** in the following way (See Fig. 2):

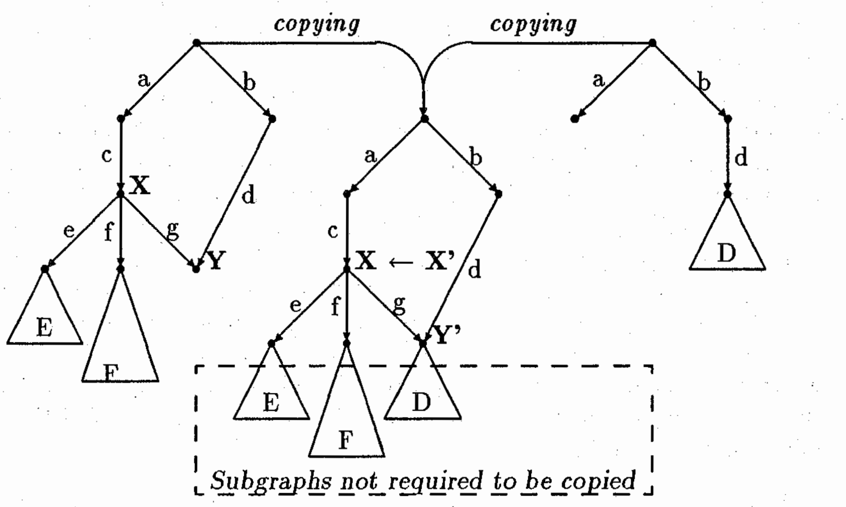
1. if both input nodes are current, the output node is one of the input nodes and is the **FORWARD** slot value of the other input node;
2. if one of input nodes is current, the output node is the current node, and is the **COPY** slot value of the other input node; or

[6] Type symbols construct a lattice

NODE	
TSYMBOL	<type symbol>
ARCS	<a list of ARC structures>
FORWARD	<a NODE structure or NIL>
COPY	<a NODE structure or NIL>
GENERATION	<an integer>

ARC	
LABEL	<a symbol>
VALUE	<a NODE structure>

Figure 1: Data Structures for Wroblewski's method



(a) *input graph G1* (c) *output graph G3* (b) *input graph G2*

Figure 2: Incremental Copy Graph Unification

3. otherwise, a new node is created as the output node. Both input node have the output node as their **FORWARD** values.

Next, the procedure then treats arcs[7]. by first treating arc pairs whose labels exist in both input nodes. The procedure applies itself recursively to each such arc pair values and adds an arc with the unification result of their values to the output node. Next, the procedure treats arcs with labels that are unique to an input node with respect to each other. Each arc value is copied and the arc with the copied value is added to the output node. For example, the node specified by the feature path $\langle a \rangle$ from input graph $G1$ ($G1/\langle a \rangle$) has an arc with the label c and the corresponding node of input graph $G2$ (i.e., $G2/\langle a \rangle$) does not. The whole subgraph rooted by $X(G1/\langle a\ c \rangle)$ is then copied. This is because such subgraphs can be modified later. For example, the $G3/\langle a\ c\ g \rangle$ node will be the copy of subgraph D.

The problem with Wroblewski's method is that the whole result DG is created by using only newly created structures. In the case of the example, the subgraphs of the result structure surrounded by the dashed rectangle can be shared with subgraphs of input structures $G1$ and $G2$. In Section 3, a method that avoids this problem is proposed.

Wroblewski's method first treats arcs with labels that exist in both input nodes and then treats arcs with unique labels. This order is related to unification failure tendency. Unification can fail in treating arcs with common labels but not in treating arcs with unique labels unless the output structure has cyclic structures. Finding failure can stop further computation as previously described, and thus finding failure first reduces unncessary computation. This order strategy can be generalized to the EFF mentioned previously and applied to the ordering of arcs with common labels. In Section 4, a method which uses this generalized strategy is proposed.

[7]The procedure assumes the existence of two procedures, namely, SharedArcs and ComplementArcs. The SharedArcs procedure takes two lists of arcs as its arguments and gives two lists of arcs each of which contains arcs whose labels exists in both lists with the same arc label order. The ComplementArcs procedure takes two lists of arcs as its arguments and gives a set of arcs whose label is unique to an input set with respect to the other.

Incremental Copy Unification

```
PROCEDURE Unify(node1, node2)
  node1 = Dereference(node1)
  node2 = Dereference(node2)
  meet = TsymbolMeet(node1.tsymbol, node2.tsymbol)
  IF Equal(meet, Bottom) THEN
    return(Bottom)
  ELSE
    outnode = GetOutNode(node1, node2, meet)
    (shareds1, shareds2) = SharedArcs(node1, node2)
    complements1 = ComplementArcs(node1, node2)
    complements2 = ComplementArcs(node2, node1)
    FOR ALL (shared1, shared2) in (shareds1, shareds2) DO
      arcnode = Unify(shared1.value, shared2.value)
      IF Equal(arcnode, Bottom) THEN
        return(Bottom)
      ELSE
        AddArc(outnode, shared1.label, arcnode)
      ENDIF
    IF Eq(outnode, node1) THEN
      complements = complement2
    ELSE IF Eq(outnode, node2) THEN
      complements = complement1
    ELSE
      complements = UnionArcs(complement1, complement2)
    ENDIF
    FOR ALL coplement IN complements DO
      newnode = CopyNode(complement.value)
      AddArc(outnode, complement.label, newnode)
  ENDIF
ENDPROCEDURE
```

Figure 3: TFS unification procedure based on Wroblewski's method (1)

Dereferencing

```
PROCEDURE Dereference(node)
  IF Node?(node.forward) THEN
    return(Dereference(node.forward))
  ELSE IF CurrrentNode?(node.copy) THEN
    return(Dereference(node.copy))
  ELSE
    return(node)
  ENDIF
ENDPROCEDURE
```

Figure 4: TFS unification procedure based on Wroblewski's method (2)

Getting Output Node

```
PROCEDURE GetOutNode(node1, node2, meet)
  IF CurrentNode?(node1) & CurrentNode?(node2) THEN
    outnode = node2
    outnode.tsymbol = meet
    node1.forward = node2
  ELSE IF CurrentNode?(node1) THEN
    outnode = node1
    outnode.tsymbol = meet
    node2.copy = node1
  ELSE IF CurrentNode?(node2) THEN
    outnode = node2
    outnode.tsymbol = meet
    node1.copy = node2
  ELSE
    outnode = CreateNode(meet)
    node1.copy = outnode
    node2.copy = outnode
  ENDIF
  return(outnode)
ENDPROCEDURE
```

Figure 5: TFS unification procedure based on Wroblewski's method (3)

3 The Lazy Incremental Copy Graph Unification Method

In Wroblewski's method, copying unique label arc values whole in order to treat cases like Fig. 2 disables structure sharing. However, this whole copying is not necessary if a lazy evaluation method is used. With this method, it is possible to delay copying a node until either its own contents need to change (e.g., node Y in Fig. 2) or until it is found to have an arc to a node that needs to be copied (e.g., node X in Fig. 2 due to change of node Y's contents). To achieve this, the LING unification method which uses copy dependency information has been developed.

The LING unification procedure uses a revised CopyNode procedure (the CopyNodeLING procedure) which does not copy structures immediately. The revised procedure uses a newly introduced slot **COPY-DEPENDENCY**. The slot has its value pairs of nodes and arcs. The revised procedure takes as its arguments the node to be copied[8] *node1*[9], the arc *arc1* whose value is the node *node1* and the mother node *node2*[10] which has the arc, and then returns the following value:

1. if the dereference result node is current, it returns the dereference result *node1'* and the arc *arc1* pair to indicate that immediately copying is necessary;
2. otherwise, the procedure adds the mother *node2* and the arc *arc1* pair into the node *node1*'s **COPY-DEPENDENCY** slot. It then recursively applies itself to each arc value with the node as the new mother node. If the recursive application returns a non-**NIL** value for several arcs, the node is copied immediately and the procedure returns the newly copied node and the arc. If it doesn't, the procedure returns **NIL**.

When a new copy of a node is required, the LING unification procedure copies structures according to the **COPY-DEPENDENCY** slot value of the node. That is, mother nodes in the **COPY-DEPENDENCY** are also copied (See the definition of the ParcolateCopy procedure).

In the above explanation, both **COPY-DEPENDENCY** and **COPY** slots are used for the sake of simplicity. However, this method can be achieved with only the **COPY** slot because a node does not have non-**NIL** **COPY-DEPENDENCY** and **COPY** slot values simultaneously.

The data in the **COPY-DEPENDENCY** slot are temporary and are discarded during an extensive process such as analyzing very ambiguous structures. However, this does not result in any incompleteness or in any partial analysis

[8] Strictly, the node which may have to be copied later
[9] an arc's destination node
[10] an arc's departure node

NODE	
TSYMBOL	<type symbol>
ARCS	<a list of ARC structures>
FORWARD	<a NODE structure or NIL>
COPY	<a NODE structure, (<an integer> . <a list of ANCESTOR structures>) or NIL>
GENERATION	<an integer>

ARC	
LABEL	<a symbol>
VALUE	<a NODE structure>

ANCESTOR	
NODE	<a NODE structure>
ARC	<an ARC structure>

Figure 6: Data Structures for the LING method

Incremental Copy Unification (LING method)

```
PROCEDURE UnifyLING(node1, node2)
  node1 = DereferenceLING(node1)
  node2 = DereferenceLING(node2)
  meet = TsymbolMeet(node1.tsymbol, node2.tsymbol)
  IF Equal(meet, Bottom) THEN
    return(Bottom)
  ELSE
    outnode = GetOutNodeLING(node1, node2, meet)
    (shareds1, shareds2) = SharedArcs(node1, node2)
    complements1 = ComplementArcs(node1, node2)
    complements2 = ComplementArcs(node2, node1)
    FOR ALL (shared1, shared2) in (shareds1, shareds2) DO
      arcnode = UnifyLING(shared1.value, shared2.value)
      IF Equal(arcnode, Bottom) THEN
        return(Bottom)
      ELSE
        AddArc(outnode, shared1.label, arcnode)
      ENDIF
    IF Eq(outnode, node1) THEN
      complements = complement2
    ELSE IF Eq(outnode, node2) THEN
      complements = complement1
    ELSE
      complements = UnionArcs(complement1, complement2)
    ENDIF
    FOR ALL coplement IN complements DO
      newnode = CopyNodeLING(compelement.value)
      AddArc(outnode, complement.label, newnode)
  ENDIF
ENDPROCEDURE
```

Figure 7: TFS unification procedure based on the LING method (1)

Dereferencing

```
PROCEDURE DereferenceLING(node)
  IF Node?(node.forward) THEN
    return(DereferenceLING(node.forward))
  ELSE IF CurrentNode?(node.copy) THEN
    return(DereferenceLING(node.copy))
  ELSE
    return(node)
  ENDIF
ENDPROCEDURE
```

Figure 8: TFS unification procedure based on the LING method (2)

structures being lost[11]. Moreover, data can be accessed in a constant order time relative to the number of DG nodes and need not be reconstructed because this method does not use data structures consisting of skeleton and environments as does Pereira's method.

The efficiency of the LING method depends on the proportion of newly created nodes in the result structures. The following worst cases can be considered.

1. If there are no arcs whose labels are unique to an input node with respect to each other, the procedure in the LING method behaves in the same way as the procedure in Wroblewski's method.

2. In the worst cases in which there are unique label arcs but all result nodes are newly created, the method has the disadvantage of treating **COPY-DEPENDENCY** slot data.

However, such cases are very rare. Usually, the number of features which exists in two input FSs is relatively small and the sizes of two input FSs are often very different. For example,

1. In Kasper's disjunctive feature description unification, a definite part FS is usually much larger than a disjunct definite part FS.

2. In sentence analysis based on HPSG or JPSG, a head constituent FS of a PP–VP complement-head construction is usually much larger than a

[11] Without structure sharing, analyzing ambiguous or long sentences by using a tabular parsing algorithm such as active chart parsing causes frequent garbage collection and page swapping occurrences. In order to avoid this, prunning partial structure candidates can be adopted. However, the adoption introduces incompleteness.

Getting Output Node for the LING method

```
PROCEDURE GetOutNodeLING(node1, node2, meet)
  IF CurrentNode?(node1) & CurrentNode?(node2) THEN
    outnode = node2
    outnode.tsymbol = meet
    node1.forward = node2
  ELSE IF CurrentNode?(node1) THEN
    outnode = node1
    outnode.copy = node2.copy
    outnode.tsymbol = meet
    node2.copy = node1
    ParcolateCopy(outnode)
  ELSE IF CurrentNode?(node2) THEN
    outnode = node2
    outnode.copy = node1.copy
    outnode.tsymbol = meet
    node1.copy = node2
    ParcolateCopy(outnode)
  ELSE
    outnode = CreateNode(meet)
    outnode.copy = Append(node1.copy, Rest(node2.copy))
    node1.copy = outnode
    node2.copy = outnode
    ParcolateCopy(outnode)
  ENDIF
  return(outnode)
ENDPROCEDURE
```

Figure 9: TFS unification procedure based on the LING method (3)

Copying Ancestors

```
PROCEDURE ParcolateCopy(node)
  IF HasCurrentAncestors?(node) THEN
    FOR ALL ancestor in Rest(node.copy) DO
      IF ancestor.node is current THEN
        newarc = CreateArc(ancestor.arc.label, node)
        ReplaceArc(ancestor.node, newarc)
      ELSE
        newnode = CreateNode(ancestor.node.tsymbol)
        newnode.copy = ancestor.node.copy
        ancestor.node.copy = newnode
        CopyArcs(ancestor.node, newnode)
        ReplaceArc(newnode, ancestor.arc)
        ParcolateCopy(newnode)
      ENDIF
  ENDIF
ENDPROCEDURE
```

Figure 10: TFS unification procedure based on the LING method (4)

Copying Node

```
PROCEDURE CopyNodeLING(node callers)
  node = Dereference(node)
  IF node is current THEN
    return(node)
  ELSE IF node is a member of set {X.node | X ∈ callers} THEN
    return(NIL)
  ELSE IF NotEmpty?(newarcs = ArcsToBeCopied(node, callers)) THEN
    newnode = CreateNode(node.tsymbol)
    node.copy = newnode
    FOR ALL arc IN node.arcs DO
      IF NotNIL?(newarc = FindArc(arc.label, newarcs)) THEN
        AddArc(newnode, newarc)
      ELSE
        AddArc(newnode, arc)
      ENDIF
    return(newnode)
  ELSE
    RegisterAncestor(node, First(callers))
  ENDIF
ENDPROCEDURE
```

Figure 11: TFS unification procedure based on the LING method (5)

Finding Arcs to Be Copied

```
PROCEDURE ArcsToBeCopeid(node callers)
  newarcs = NIL
  FOR ALL arc IN node.arcs DO
    newcallers = Cons(CreateAncestor(node, arc), callers)
    newnode = CopyNodeLING(arc.value, newcallers)
    IF NotNIL?(newnode) THEN
      newarc = CreateArc(arc.label, newnode)
      newarcs = {newarc} ∪ newarcs
    ENDIF
  return(newarcs)
ENDPROCEDURE
```

Figure 12: TFS unification procedure based on the LING method (6)

complement constituent FS[12], and a complement constituent FS of a VP–AUXV complement-head construction is usually much larger than a head constituent FS[13].

To compare the efficiencies of Wroblewski's method and the LING method, brief experiments have been applied. TFSs like the followings have been applied both to Wroblewski's method and the LING method.

To compare efficiencies of Wroblewski's method and the LING method, TFSs like the followings are applied to both methods:

[12] E.g., jimukyoku ni (complement) + tourokuyoushi o shikyuu okuru (head)

[13] E.g., jimukyoku ni tourokuyoushi o shikyuu okura (complement) + seru (head)

$$G_{in1} = \mathbf{TOP} \begin{bmatrix} \mathrm{E} & \mathbf{TOP} \begin{bmatrix} \mathrm{F} & {}_{?\mathrm{TOP\_001}}\mathbf{TOP} \begin{bmatrix} 2 & \mathbf{TOP} \begin{bmatrix} 0 & \mathbf{0} \\ 1 & \mathbf{1} \\ 2 & \mathbf{2} \end{bmatrix} \\ 0 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 0 & \mathbf{0} \\ 2 & \mathbf{2} \end{bmatrix} \\ 1 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 2 & \mathbf{2} \\ 0 & \mathbf{0} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathrm{A} & \mathbf{TOP} \begin{bmatrix} \mathrm{B} & \mathbf{TOP} \begin{bmatrix} \mathrm{D} & \mathbf{TOP} \begin{bmatrix} 2 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 0 & \mathbf{0} \\ 2 & \mathbf{2} \end{bmatrix} \\ 0 & \mathbf{TOP} \begin{bmatrix} 0 & \mathbf{0} \\ 1 & \mathbf{1} \\ 2 & \mathbf{2} \end{bmatrix} \\ 1 & \mathbf{TOP} \begin{bmatrix} 0 & \mathbf{0} \\ 2 & \mathbf{2} \\ 1 & \mathbf{1} \end{bmatrix} \end{bmatrix} \\ \mathrm{C} & {}_{?\mathrm{TOP\_001}} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$
G_{in2} = \mathbf{TOP} \begin{bmatrix}
\mathrm{H} & \mathbf{TOP} \begin{bmatrix} \mathrm{I} & \mathbf{TOP} \begin{bmatrix}
2 & \mathbf{TOP} \begin{bmatrix} 0 & \mathbf{0} \\ 2 & \mathbf{2} \\ 1 & \mathbf{1} \end{bmatrix} \\
1 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 2 & \mathbf{2} \\ 0 & \mathbf{0} \end{bmatrix} \\
0 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 0 & \mathbf{0} \\ 2 & \mathbf{2} \end{bmatrix}
\end{bmatrix} \end{bmatrix} \\
\mathrm{E} & \mathbf{TOP} \begin{bmatrix} \mathrm{F} & \mathbf{TOP} \begin{bmatrix}
2 & \mathbf{TOP} \begin{bmatrix} 2 & \mathbf{2} \\ 1 & \mathbf{1} \\ 0 & \mathbf{0} \end{bmatrix} \\
1 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 0 & \mathbf{0} \\ 2 & \mathbf{2} \end{bmatrix} \\
0 & \mathbf{TOP} \begin{bmatrix} 2 & \mathbf{2} \\ 0 & \mathbf{0} \\ 1 & \mathbf{1} \end{bmatrix}
\end{bmatrix} \end{bmatrix} \\
\mathrm{A} & \mathbf{TOP} \begin{bmatrix} \mathrm{G} & \mathbf{TOP} \begin{bmatrix}
2 & \mathbf{TOP} \begin{bmatrix} 1 & \mathbf{1} \\ 0 & \mathbf{0} \\ 2 & \mathbf{2} \end{bmatrix} \\
1 & \mathbf{TOP} \begin{bmatrix} 2 & \mathbf{2} \\ 1 & \mathbf{1} \\ 0 & \mathbf{0} \end{bmatrix} \\
0 & \mathbf{TOP} \begin{bmatrix} 0 & \mathbf{0} \\ 1 & \mathbf{1} \\ 2 & \mathbf{2} \end{bmatrix}
\end{bmatrix} \end{bmatrix}
\end{bmatrix}
$$

$$
G_{out} = \mathbf{TOP}\left[\begin{array}{ll}
\mathrm{H} & \mathbf{TOP}\left[\begin{array}{ll}\mathrm{I} & \mathbf{TOP}\left[\begin{array}{ll}
2 & \mathbf{TOP}\begin{bmatrix}0&0\\2&2\\1&1\end{bmatrix}\\
1 & \mathbf{TOP}\begin{bmatrix}1&1\\2&2\\0&0\end{bmatrix}\\
0 & \mathbf{TOP}\begin{bmatrix}1&1\\0&0\\2&2\end{bmatrix}
\end{array}\right]\end{array}\right]\\
\mathrm{E} & \mathbf{TOP}\left[\begin{array}{ll}\mathrm{F} & {}^{?\mathrm{TOP\_001}}\mathbf{TOP}\left[\begin{array}{ll}
2 & \mathbf{TOP}\begin{bmatrix}0&0\\1&1\\2&2\end{bmatrix}\\
0 & \mathbf{TOP}\begin{bmatrix}1&1\\0&0\\2&2\end{bmatrix}\\
1 & \mathbf{TOP}\begin{bmatrix}1&1\\2&2\\0&0\end{bmatrix}
\end{array}\right]\end{array}\right]\\
\mathrm{A} & \mathbf{TOP}\left[\begin{array}{ll}
\mathrm{B} & \mathbf{TOP}\left[\begin{array}{ll}
\mathrm{D} & \mathbf{TOP}\left[\begin{array}{ll}
2 & \mathbf{TOP}\begin{bmatrix}1&1\\0&0\\2&2\end{bmatrix}\\
0 & \mathbf{TOP}\begin{bmatrix}0&0\\1&1\\2&2\end{bmatrix}\\
1 & \mathbf{TOP}\begin{bmatrix}0&0\\2&2\\1&1\end{bmatrix}
\end{array}\right]\\
\mathrm{C} & ?\mathrm{TOP\_001}
\end{array}\right]\\
\mathrm{G} & \mathbf{TOP}\left[\begin{array}{ll}
2 & \mathbf{TOP}\begin{bmatrix}1&1\\0&0\\2&2\end{bmatrix}\\
1 & \mathbf{TOP}\begin{bmatrix}2&2\\1&1\\0&0\end{bmatrix}\\
0 & \mathbf{TOP}\begin{bmatrix}0&0\\1&1\\2&2\end{bmatrix}
\end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Table 1: Comparison of Wroblewski's and the LING method

	Elapsed Time (sec)	
Number of result nodes	Wroblewski's method	LING method
44	0.015	0.014
125	0.112	0.035
368	0.540	0.249
1461	1.782	0.656
4377	5.790	2.038
13125	43.025	8.102

Experiment results are summarized in Table. 1 and Fig. 3. Each elapsed time is the average of 50 evaluations in Symbolics Common Lisp. Elaspsed times are aproximated by the following formulas;

$$T_{Wroblewski} = 1.60 \times 10^{-4} N^{1.29}$$
$$T_{LING} = 2.25 \times 10^{-4} N^{1.10}$$

where N is the number of result nodes. These results indicate that the LING method is more efficient than Wroblewski's method when result FSs are large.

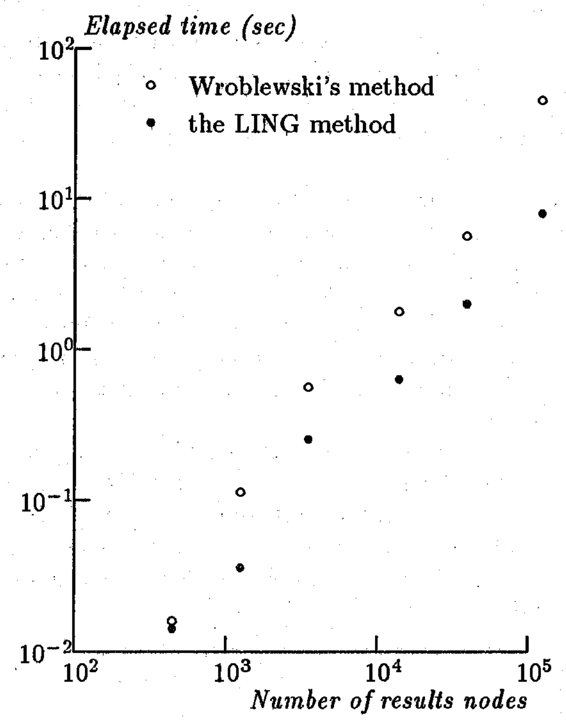


Figure 13: Comparison of Wroblewski's and the LING method

4 The Strategic Incremental Copy Graph Unification Method

In certain processes where FS unification is applied, there are features whose values fail in unification with other values relatively often and there are features whose values do not fail so often. For example, in Japanese analysis, unification of features for conjugation forms, case markers, and sortal restrictions[14] tends to fail but unification of features for semantic representations[15] does not. In such cases, application of the EFF strategy, that is, treating features tending to fail in unification first, reduces unncessary computation when the unification finally fails. For example, when unification of features for case markers does fail, treating these features first avoids treating features for semantic representations. The strategic incremental copy graph unification (SING unification) method uses this failure tendency.

These unification failure tendencies depend on a process within which FS unification is applied, such as analysis or generation. Contrary to the analysis case, unification of features for semantic representation tends to fail. In this method, therefore, the failure tendency information is aquired by a learning process. That is, the SING unification procedure applied in an analysis process uses the failure tendency information acquired by a learning analysis process, and the procedure applied in a generation process uses the information acquired by a learning generation process.

In the learning process, when FS unification is applied, feature treatment orders are randomized for the sake of random extraction. As in TFS unification, failure tendency information is recorded in terms of triple consisting of the most generic specifier type symbol of the input TFSs' type symbols, a feature, and sucess/failure flag. This is because the type symbol of a TFS represents salient information on the whole TFS.

By using learned failure tendency information, feature value unification is applied in an order that first treats features which tend to fail. This is achieved by sorting shared arc pairs.

The efficiency of the SING method depends on the following factors:

The overall unification failure rate of the process: in extreme cases, if no unification failure occurs, the method has no advantages except the overhead of feature unification order sorting. However, such cases do not occur in practice.

Number of features feature structures have: if each feature structure has only a small number of features, the efficiency gain from the SING method is small.

[14] In the current NADINE grammar, CFORM, FORM and SEMF features

[15] SEM feature

Unevenness of feature unification failure tendency: in extreme cases, if every feature has the same failure tendency, this method has no advantage. However, such cases are very rare, and for example, in many cases of natural language analysis, FS unification failures occur in treating only limited kinds of features related to grammatical agreement such as number/person agreement and case marker agreement and semantic selectional constraints. In such cases, the SING method is efficient.

The above factors can be examined by inspecting acquired failure tendency information, from which the efficiency gain from the SING method can be predicted. Moreover, it is possible for each type symbol to select whether to apply feature unification order sorting or not.

Strategic Incremental Copy Unification (SING method)

```
PROCEDURE UnifySING(node1, node2)
  node1 = DereferenceLING(node1)
  node2 = DereferenceLING(node2)
  meet = TsymbolMeet(node1.tsymbol, node2.tsymbol)
  IF Equal(meet, Bottom) THEN
    return(Bottom)
  ELSE
    outnode = GetOutNode(node1, node2, meet)
    (shareds1, shareds2) = SharedArcs(node1, node2)
    complements1 = ComplementArcs(node1, node2)
    complements2 = ComplementArcs(node2, node1)
    (shareds1, shareds2)
       = SortArcPairs(meet, shareds1, shareds2)
    FOR ALL (shared1, shared2) in (shareds1, shareds2) DO
      arcnode = UnifySING(shared1.value, shared2.value)
      IF Equal(arcnode, Bottom) THEN
        RegisterTendency(meet, shareds1.label, FAILURE)
        return(Bottom)
      ELSE
        RegisterTendency(meet, shareds1.label, SUCCESS)
        AddArc(outnode, shared1.label, arcnode)
      ENDIF
    IF Eq(outnode, node1) THEN
      complements = complement2
    ELSE IF Eq(outnode, node2) THEN
      complements = complement1
    ELSE
      complements = UnionArcs(complement1, complement2)
    ENDIF
    FOR ALL coplement IN complements DO
      newnode = CopyNode(compelement.value)
      AddArc(outnode, complement.label, newnode)
  ENDIF
ENDPROCEDURE
```

Figure 14: TFS unification procedure based on the SING method

5 Conclusion

This technical report proposes two incremental copy graph unification methods, the Lazy Incremental Copy Graph unification method and the Strategic Incremental Copy Graph unification method. The LING unification method achieves structure sharing without the ($\log d$) data access overhead of Pereira's method. Structure sharing avoids wasting memory. Furthermore, structure sharing increases the porpotion of token identical substructures of FSs which makes keeping unification results of substructures of FSs and reusing them efficiently. This reduces repeated calculation of substructures.

The SING unification method introduces the concept of feature unification strategy. These two unification methods can be combined and the combined method is called the Strategic Lazy Incremental Copy Graph unification method. The combined method not only makes each FS unification efficient but also reduces garbage collection and page swapping occurrences, thus increasing total efficiencies of TFS unification-based natural language processing systems such as a spoken Japanese sentence analysis system based on HPSG.

References

[1] K. Kogure et al. A method of analyzing Japanese speech act types. In *The Proceedings of the 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Carnegie Mellon University, Pittsuburgh, PA, 1988.

[2] G. K. Pullum G. Gazder, E. Klein and I. A. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, 1985.

[3] G. Gazder. Phrase structure grammar. In P. Jacobson and K. Pullum, editors, *The Nature of Syntactic Representations*, D. Reidel, 1982.

[4] T. Gunji. *Japanese Phrase Structure Grammar*. D. Reidel, 1987.

[5] R. Kaplan and J. Bresnan. Lexical Functional Grammar: a formal sys-tem for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, 1982.

[6] L. Karttunen. *D-PATR — A Development Environment for Unification-Based Grammars*. Technical Report CSLI-86-61, CSLI, 1986.

[7] R. T. Kasper. A unification method for disjunctive feature descriptions. In *The Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 235–242, The Association for Computational Linguistics, Stanford University, Stanford, CA, July 1987.

[8] K. Kogure. Parsing Japanese spoken sentences based on HPSG. In *The Proceedings of the International Workshop on Parsing Technologies*, pages 132–141, Carnegie Mellon University, Pittsuburgh, PA, 1989.

[9] F. C. N. Pereira. Structure sharing representation for unification-based formalisms. In *The Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, The Association for Computational Linguistics, University of Chicago, Chicago, IL, 1985.

[10] C. Pollard. *Generalized Context-Free Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.

[11] C. Pollard. *An Information-Based Syntax and Semantics — Volume 1: Fundamentals. CSLI Lecture Note Number 13*, CSLI, 1987.

[12] D. Wroblewski. Nondestructive graph unification. In *The Proceedings of the 6th National Conference on Artificial Intelligence*, AAAI, 1987.