Strategic Lazy Incremental Copy Graph Unification

Kiyoshi KOGURE†

ATR Interpreting Telephony Research Laboratories
Sanpeidani Inuidani, Seika-cho, Soraku-gun, Kyoto 619-02, Japan
kogure%atom.ntt.jp@relay.cs.net

Abstract

The strategic lazy incremental copy graph unification method is a combination of two methods for unifying feature structures. One, called the lazy incremental copy graph unification method, achieves structure sharing with constant order data access time which reduces the required memory. The other, called the strategic incremental copy graph unification method, uses an early failure finding strategy which first tries to unify substructures tending to fail in unification; this method is based on stochastic data on the likelihood of failure and reduces unnecessary computation. The combined method makes each feature structure unification efficient and also reduces garbage collection and page swapping occurrences, thus increasing the total efficiency of natural language processing systems mainly based on typed feature structure unification such as natural language analysis and generation systems.

1. Introduction

Various kinds of grammatical formalisms without transformation were proposed from the late 1970s through the 1980s [Gazder et al 85, Kaplan and Bresnan 82, Kay 85, Pollard and Sag 87]. These formalisms were developed relatively independently but actually had common properties; that is, they used data structures called functional structures or feature structures and they were based on unification operation on these data structures. These formalisms were applied in the field of natural language processing and, based on these formalisms, systems such as machine translation systems were developed [Kogure et al 88].

In such unification-based formalisms, feature structure (FS) unification is the most fundamental and significant operation. The efficiency of systems based on such formalisms, such as natural language analysis and generation systems very much depends on their FS unification efficiencies. This dependency is especially crucial for lexicon-driven approaches such as HPSG[Pollard and Sag 86] and JPSG[Gunji 87] because rich lexical information and phrase structure information is described in terms of FSs. For example, a spoken Japanese analysis system based on HPSG[Kogure 89] uses 90% - 98% of the elapsed time in FS unification.

Several FS unification methods were proposed in [Karttunen 86, Pereira 85, Wroblewski 87]. These methods uses rooted directed graphs (DGs) to represent FSs. These methods take two DGs as their inputs and give a unification result DG. Previous research identified DG copying as a significant overhead. Wroblewski claims that copying is wrong when an algorithm copies too much (*over copying*) or copies too soon (*early copying*). He proposed an incremental copy graph unification method to avoid *over copying* and *early copying*.

However, the problem with his method is that a unification result graph consists only of newly created structures. This is unnecessary because there are often input subgraphs that can be used as part of the result graph without any modification, or as sharable parts between one of the input graphs and the result graph. Copying sharable parts is called *redundant copying*. A better method would minimize the copying of sharable parts. The redundantly copied parts are relatively large when input graphs have few common feature paths. In natural language processing, such cases are ubiquitous. For example, in unifying an FS representing constraints on phrase structures and an FS representing a daughter phrase structure, such cases occur very frequently. In Kasper's disjunctive feature description unification [Kasper 86], such cases occur very frequently in unifying definite and disjunct's definite parts. Memory is wasted by such redundant copying and this causes frequent garbage collection and page swapping which decrease the total system efficiency. Developing a method which avoids memory wastage is very important.

Pereira's structure sharing FS unification method can avoid this problem. The method achieves structure sharing by importing the Boyer and Moore approach for term structures[Boyer and Moore 72]. The method uses a data structure consisting of a skeleton part to represent original information and an environment part to represent updated information. The skeleton part is shared by one of the input FSs and the result FS. Therefore, Pereira's method needs relatively few new structures when two input FSs are difference in size and which input is larger are known before unification.

However, Pereira's method can create skeleton-environment structures that are deeply embedded, for example, in recursively constructing large phrase structure from their parts. This causes *O(log d)* graph node access time overhead in assembling the whole DG

† Present affiliation: Information Science Research Laboratory, NTT Basic Research Laboratories.
Present address: 9-11, Midori-cho 3-chome, Musashino-shi, Tokyo 180, Japan.

from the skeleton and environments where d is the number of nodes in the DG. Avoiding this problem in his method requires a special operation of merging a skeleton-environment structure into a skeleton structure, but this prevents structure sharing.

This paper proposes an FS unification method that allows structure sharing with constant order node access time. This method achieves structure sharing by introducing lazy copying to Wroblewski's incremental copy graph unification method. The method is called the *lazy incremental copy graph* unification method (the LING unification method for short).

In a natural language processing system that uses declarative constraint rules in terms of FSs, FS unification provides constraint-checking and structure-building mechanisms. The advantages of such a system 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., 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 phrase structure from its parts (e.g., a sentence from a subject NP and VP), unnecessary computation can be reduced if the semantic representation is assembled after checking constraints such as grammatical agreements, which can fail. This is impossible in straightforward unification-based formalisms.

In contrast, in a procedure-based system which uses IF-THEN style rules (i.e., consisting of explicit test and structure-building operations), it is possible to construct the semantic representation (THEN part) after checking the agreement (IF part). Such a system has the advantage of processing efficiency but the disadvantage of lacking multi-directionality.

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 that 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 specified explicitly by rule writers, but are controlled by learned information on tendencies of FS constraint application failures. This method is called the *strategic incremental copy graph* unification method (the SING unification method).

These two methods can be combined into a single method called the *strategic lazy incremental copy graph* unification method (the SLING unification method).

Section 2 explains typed feature structures (TFSs) and unification on them. Section 3 explains a 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. Section 3 and 4 introduce the LING method and the SING method, respectively.

2. Typed Feature Structures

Ordinary FSs used in unification-based grammar formalisms such as PATR[Shieber 85] are classified into two classes, namely, atomic FSs and complex FSs. An atomic FS is represented by an atomic symbol and a complex FS is represented by a set of feature-value pairs. Complex FSs are used to partially describe objects by specifying values for certain features or attributes of described objects. Complex FSs can have complex FSs as their feature values and can share certain values among features. For ordinary FSs, unification is defined by using partial ordering based on subsumption relationships. These properties enable flexible descriptions.

An extension allows complex FSs to have type symbols which define a lattice structure on them, for example, as in [Pollard and Sag 87]. The type symbol lattice contains the greatest type symbol Top, which subsumes every type symbol, and the least type symbol Bottom, which is subsumed by every type symbol. An example of a type symbol lattice is shown in Fig. 1.

An extended complex FS is represented by a type symbol and a set of feature-value pairs. Once complex FSs are extended as above, an atomic FS can be seen as an extended complex FS whose type symbol has only Top as its greater type symbol and only Bottom as its lesser type symbol and which has an empty set of feature value pairs. Extended complex FSs are called typed feature structures (TFSs). TFSs are denoted by feature-value pair matrices or rooted directed graphs as shown in Fig. 2.

Among such structures, unification can be defined [Aït-Kaci 86] by using the following order;

A TFS $t1$ is less than or equal to a TFS $t2$ if and only if:

- the type symbol of $t1$ is less than or equal to the type symbol of $t2$; and
- each of the features of $t2$ exists in $t1$ and has as its value a TFS which is not less than its counterpart in $t1$; and
- each of the coreference relationships in $t2$ is also held in $t1$.

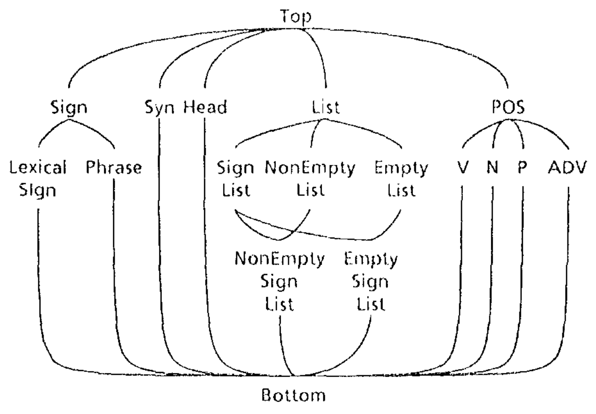
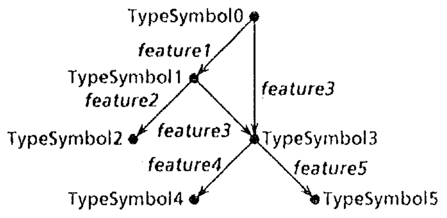


Figure 1: Example of a type symbol lattice

```
TypeSymbol0
[feature1  TypeSymbol1
           [feature2  TypeSymbol2
            feature3  ?Tag  TypeSymbol3
                            [feature4  TypeSymbol4
                             feature5  TypeSymbol5 ]]
 feature3  ?Tag                                     ]
```

(a) feature-value matrix notation
"?" is the prefix for a tag and TFSs with the same tag are token-identical.



(b) directed graph notation

Figure 2: TFS notations

```
Phrase
[syn   Syn
       [head    ?X1 Head
        subcat  ?X2 SignList ]
 dtrs  CHconst
       [cdtr  ?X3 Sign
        hdtr  Sign
              [syn  Syn
                    [head    ?X1
                     subcat  NonEmptySignList
                             [first  ?X3
                              rest   ?X2    ]]]]]

                                                   ⊔

Phrase
[dtrs  CHconst
       [hdtr  LexicalSign
              [syn  Syn
                    [head    Head
                             [pos   P
                              form  Ga ]
                     subcat  NonEmptySignList
                             [first  Sign
                                     [syn  Syn
                                           [head  Head
                                                  [pos  N]]]
                              rest   EmptySignList           ]]]]]

   Phrase
   [syn   Syn
          [head    ?X1 Head
                       [pos   P
                        form  Ga ]
           subcat  ?X2 EmptySignList ]
    dtrs  CHconst
=         [cdtr  ?X3 Sign
                 [syn  Syn
                       [head  Head
                              [pos  N ]]]
           hdtr  LexicalSign
                 [syn  Syn
                       [head    ?X1
                        subcat  NonEmptySignList
                                [first  ?X3
                                 rest   ?X2    ]]]]]
```

Figure 3: Example of TFS unification

Then, the unification of *t1* and *t2* is defined as their greatest lower bound or the meet. A unification example is shown in Fig. 3. In the directed graph notation, TFS unification corresponds to graph merging. TFSs are very convenient for describing linguistic information in unification-based formalisms.

3. Wroblewski's Incremental Copy Graph Unification Method and Its Problems

In TFS unification based on Wroblewski's method, a DG is represented by the NODE and ARC structures corresponding to a TFS and a feature-value pair respectively, as shown in Fig. 4. The NODE structure has the slots TYPESYMBOL 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. When a NODE's GENERATION value is equal to the global value specifying the current unification process, the structure has been created in the current process or that the structure is *current*.

The characteristics which allow nondestructive incremental copy are the NODE's two different slots, FORWARD and COPY, for representing forwarding relationships. A FORWARD slot value represents an eternal relationship while a COPY slot value represents a temporary relationship. When a NODE *node1* has a NODE *node2* as its FORWARD value, the other contents of the *node1* are ignored and the contents of *node2* are used. However, when a NODE has another NODE as its COPY value, the contents of the COPY value are used only when the COPY value is *current*. After the process finishes, all COPY slot values are ignored and thus original structures are not destroyed.

The unification procedure based on this method takes as its input two nodes which are roots of the DGs to be unified. The procedure incrementally copies nodes and arcs on the subgraphs of each input DG until a node with an empty ARCS value is found.

The procedure first dereferences both root nodes of the input DGs (i.e., it follows up FORWARD and COPY slot values). If the dereference result nodes are identical, the procedure finishes and returns one of the dereference result nodes.

Next, the procedure calculates the meet of their type symbol. If the meet is Bottom, which means inconsistency, the procedure finishes and returns Bottom. Otherwise, the procedure obtains the output node with the meet as its TYPESYMBOL. The output node has been created only when neither input node is current; or otherwise the output node is an existing current node.

Next, the procedure treats arcs. 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

NODE	
TYPESYMBOL:	<*symbol*>
ARCS:	<*a list of* ARC *structures*>
FORWARD:	<*a* NODE *structure or NIL*>
COPY:	<*a* NODE *structure or NIL*>
GENERATION:	<*an integer*>

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

Figure 4: Data Structures for Wroblewski's method

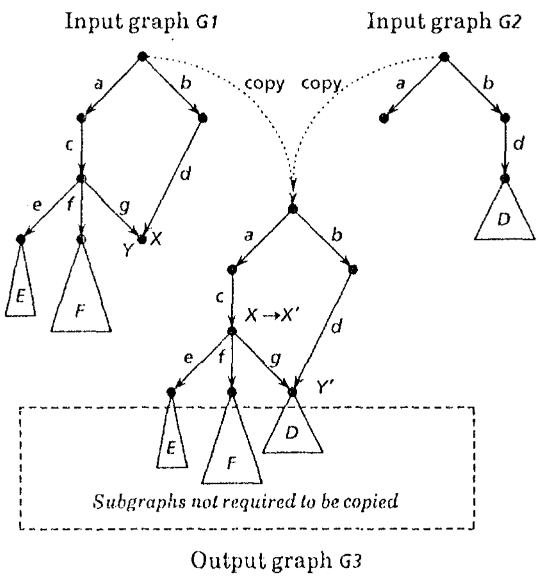


Figure 5: Incremental copy graph unification
In this figure, type symbols are omitted.

its arguments and gives one list of arcs whose labels are unique to one input list.

The unification procedure first treats arc pairs obtained by SharedArcs. The procedure applies itself recursively to each such arc pair values and adds to the output node every arc with the same label as its label and the unification result of their values unless the unification result is Bottom.

Next, the procedure treats arcs obtained by ComplementArcs. Each arc value is copied and an arc with the same label and the copied value is added to the output node. For example, consider the case when feature *a* is first treated at the root nodes of *G1* and *G2* in Fig. 5. The unification procedure is applied recursively to feature *a* values of the input nodes. The node specified by the feature path <*a*> from input graph *G1* (*G1*/<*a*>) has an arc with the label c and the corresponding node of input graph *G2* does not. The whole subgraph rooted by *G1*/<*a c*> is then copied. This is because such subgraphs can be modified later. For example, the node Y(*G3*/<*a c g*>) will be modified to be the unification result of *G1*/<*a c g*> (or *G1*/<*b d*>) and *G2*/<*b d*> when the feature path <*b d*> will be treated.

```
Incremental Copy Graph Unification

PROCEDURE Unify(node1, node2)
  node1 = Dereference(node1).
  node2 = Dereference(node2).
  IF Eq?(node1, node2) THEN
    Return(node1).
  ELSE
    meet = Meet(node1.typesymbol, node2.typesymbol)
    IF Equal?(meet, Bottom) THEN
      Return(Bottom).
    ELSE
      outnode = GetOutNode(node1, node2, meet).
      (shareds1, shareds2)
            = SharedArcs(node1.arcs, node2.arcs).
      complements1
            = ComplementArcs(node1.arcs, node2.arcs).
      complements2
            = ComplementArcs(node2.arcs, node1.arcs).
      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
          = Append(complements1, complements2).
      ENDIF
      FOR ALL complement IN complements
      DO
        newnode = CopyNode(complement.value).
        AddArc(outnode, complement.label, newnode).
      Return(outnode).
    ENDIF
  ENDIF
ENDPROCEDURE
```

Figure 6: Incremental copy graph unification procedure

The problem with Wroblewski's method is that the whole result DG is created by using only newly created structures. In the example in Fig. 5, the subgraphs of the result DG surrounded by the dashed rectangle can be shared with subgraphs of input structures *G1* and *G2*. Section 4 proposes a method that avoids this problem.

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 the unification failure tendency. Unification fails in treating arcs with common labels more often than in treating arcs with unique labels. Finding a failure can stop further computation as previously described, and thus finding failures first reduces unnecessary computation. This order strategy can be generalized to the EFF and applied to the ordering of arcs with common labels. In Section 5, a method which uses this generalized strategy is proposed.

4. The Lazy Incremental Copy Graph Unification Method

In Wroblewski's method, copying unique label arc values whole in order to treat cases like Fig. 5 disables structure sharing. However, this whole copying is not necessary if a lazy evaluation method is used. With such a method, it is possible to delay copying a node until either its own contents need to change (e.g., node *G3*/<*a c* *g*>) or until it is found to have an arc (sequence) to a node that needs to be copied (e.g., node *X G3/<a c>* in Fig. 5 due to a change of node *Y G3/<a c g>*). To achieve this, the LING unification method, which uses copy dependency information, was developed.

The LING unification procedure uses a revised CopyNode procedure which does not copy structures immediately. The revised procedure uses a newly introduced slot COPY-DEPENDENCY. The slot has pairs consisting of nodes and arcs as its value. The revised CopyNode procedure takes as its inputs the node to be copied *node1* and the arc *arc1* with *node1* as its value and *node2* as its immediate ancestor node (i.e., the arc's initial node), and does the following (see Fig. 7):

(1) if *node1'*, the dereference result of *node1*, is current, then CopyNode returns *node1'* to indicate that the ancestor node *node2* must be copied immediately;

(2) otherwise, CopyArcs is applied to *node1'* and if it returns several arc copies, CopyNode creates a new copy node. It then adds the arc copies and arcs of *node1'* that are not copied to the new node, and returns the new node;

(3) otherwise, CopyNode adds the pair consisting of the ancestor node *node2* and the arc *arc1* into the COPY-DEPENDENCY slot of *node1'* and returns *NIL*.

CopyArcs applies CopyNode to each arc value with *node1'* as the new ancestor node and returns the set of new arcs for non-*NIL* CopyNode results.

When a new copy of a node is needed later, the LING unification procedure will actually copy structures using the COPY-DEPENDENCY slot value of the node (in GetOutNode procedure in Fig. 6). It substitutes arcs with newly copied nodes for existing arcs. That is, antecedent nodes in the COPY-DEPENDENCY values are also copied.

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 values simultaneously.

The data in the COPY-DEPENDENCY slot are temporary and they are discarded during an extensive process such as analyzing a sentence. However, this does not result in any incompleteness or in any partial analysis structure being lost. 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 a data structure consisting of skeleton and environments as does Pereira's method.

The efficiency of the LING unification method depends on the proportion of newly created structures in the unification result structures. Two 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 LING unification method behaves in the same way as the procedure in the Wroblewski's method.

(2) In the worst cases, in which there are unique label arcs but all result structures are newly created, the method

```
                         CopyNode
PROCEDURE CopyNode(node, arc, ancestor)
  node = Dereference(node).
  IF Current?(node) THEN
    Return(node).
  ELSE IF NotEmpty?(newarcs = CopyArcs(node))
  THEN
    newnode = Create(node.typesymbol).
    node.copy = newnode.
    FOR ALL arc IN node.arcs DO
      IF NotNIL?(newarc = FindArc(arc.label, newarcs))
      THEN
        AddArc(newnode, newarc.label, newarc.value).
      ELSE
        AddArc(newnode, arc.label, arc.value).
      ENDIF
    Return(newnode).
  ELSE
    node.copy-dependency
      = node.copy-dependency ∪ {Cons(ancestor, arc)}.
    Return(NIL).
  ENDIF
ENDPROCEDURE
```

```
                         CopyArcs
PROCEDURE ArcsCopied(node)
  newarcs = {}.
  FOR ALL arc IN node.arcs DO
    newnode = CopyNode(arc.value, arc, node).
    IF NotNIL?(newnode) THEN
      newarc = CreateArc(arc.label, newnode).
      newarcs = {newarc} ∪ newarcs.
    ENDIF
  Return(newarcs).
ENDPROCEDURE
```

Figure 7: The revised CopyNode procedure

has the disadvantage of treating copy dependency information.

However, these two cases are very rare. Usually, the number of features in two input structures is relatively small and the sizes of the two input structures are often very different. For example, in Kasper's disjunctive feature description unification, a definite part FS is larger than a disjunct definite part FS.

5. The Strategic Incremental Copy Graph Unification Method

In a system where FS unification is applied, there are features whose values fail relatively often in unification with other values and there are features whose values do not fail so often. For example, in Japanese sentence analysis, unification of features for conjugation forms, case markers, and semantic selectional restrictions tends to fail but unification of features for semantic representations does not fail. In such cases, application of the EFF strategy, that is, treating features tending to fail in unification first, reduces unnecessary 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 SING unification method uses this failure tendency information.

These unification failure tendencies depend on systems such as analysis systems or generation systems.

Unlike the analysis case, unification of features for semantic representations tends to fail. In this method, therefore, the failure tendency information is acquired by a learning process. That is, the SING unification method applied in an analysis system uses the failure tendency information acquired by a learning analysis 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 a triplet consisting of the greatest lower bound type symbol of the input TFSs' type symbols, a feature and success/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 with the greatest tendency to fail. This is achieved by the sorting procedure of common label arc pairs attached to the meet type symbol. The arc pairs obtained by the SharedArcs procedure are sorted before treating arcs.

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

(1) The overall FS 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.

(2) Number of features FSs have: if each FS has only a small number of features, the efficiency gain from the SING unification method is small.

(3) Unevenness of FS unification failure tendency: in extreme cases, if every feature has the same unification failure tendency, this method has no advantage. However, such cases do not occur or 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 and/or person agreement and semantic selectional constraints. In such cases, the SING unification method obtains efficiency gains.

The above factors can be examined by inspecting 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.

6. Conclusion

The *strategic lazy incremental copy graph* (SLING) unification method combines two incremental copy graph unification methods: the *lazy incremental copy graph* (LING) unification method and the *strategic incremental copy graph* (SING) unification method. The LING unification method achieves structure sharing without the *O(log d)* data access overhead of Pereira's method. Structure sharing avoids memory wastage. Furthermore, structure sharing increases the portion of token identical substructures of FSs which makes it efficient to keep unification results of substructures of FSs and reuse them. This reduces repeated calculation of substructures.

The SING unification method introduces the concept of feature unification strategy. The method treats features tending to fail in unification first. Thus, the efficiency gain from this method is high when the overall FS unification failure rate of the application process is high.

The combined method makes each FS unification efficient and also reduces garbage collection and page swapping occurrences by avoiding memory wastage, thus increasing the total efficiency of FS unification-based natural language processing systems such as analysis and generation systems based on HPSG.

Acknowledgement

I would like to thank Akira Kurematsu, Hitoshi Iida, and my other colleagues both at ATR Interpreting Telephony Research Laboratories and NTT Basic Research Laboratories for their encouragement and thought-provoking discussions.

Reference

[Aït-Kaci 86] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. In *Journal of Theoretical Computer Science 45*, 1986.

[Boyer and Moore 72] R. S. Boyer and J. S. Moore. The sharing of structures in theorem-proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, 1972.

[Gazder et al 85] G. Gazder, G. K. Pullum, E. Klein and I. A. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, 1985.

[Gunji 87] T. Gunji. *Japanese Phrase Structure Grammar*. D. Reidel, 1987.

[Kaplan and Bresnan 82] R. Kaplan and J. Bresnan. Lexical Functional Grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, 1982.

[Karttunen 86] L. Karttunen. *D-PATR - A Development Environment for Unification-Based Grammars*. CSLI-86-91, CSLI, 1986.

[Kasper 87] R. T. Kasper. A unification method for disjunctive feature descriptions. In *the Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1987.

[Kay 85] M. Kay. Parsing in functional grammar. In D. Dowty, L. Karttunen and A. Zwicky, editors, *Natural Language Parsing*, Cambridge University Press, 1985.

[Kogure et al 88] 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*, 1988.

[Kogure 89] K. Kogure. Parsing Japanese spoken sentences based on HPSG. In *the Proceedings of the International Workshop on Parsing Technologies*, 1989.

[Pereira 85] 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*, 1985.

[Pollard and Sag 87] C. Pollard and I. Sag. *An Information-Based Syntax and Semantics*. CSLI Lecture Notes No. 13, CSLI, 1987.

[Shieber 86] S. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes No. 4, CSLI, 1986.

[Wroblewski 87] D. Wroblewski. Nondestructive graph unification. In *the Proceedings of the 6th National Conference on Artificial Intelligence*, 1987.