Carnegie-Mellon University
PITTSBURGH PENNSYLVANIA 1900

# Carnegie-Mellon University

*upon recommendation of its Faculty,*
*hereby confers upon*

HIDETO TOMABECHI

*the degree of*

DOCTOR OF PHILOSOPHY

*in recognition of*
*the completion of the course of study*
*prescribed for this degree*
*in the field(s) of*

COMPUTATIONAL LINGUISTICS

*Given under the seal of*
*the corporation at Pittsburgh*
*in the Commonwealth of Pennsylvania*
*on May 16, 1993*

| | |
|---|---|
| Robert A. Charpie | Robert Mehrabian |
| CHAIRMAN OF THE BOARD OF TRUSTEES | PRESIDENT |

# Efficient Unification for Natural Language

by

HIDETO TOMABECHI

Submitted to the Program in Computational Linguistics
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

at the

CARNEGIE MELLON UNIVERSITY

May 1993

# Efficient Unification for Natural Language

by

Hideto Tomabechi

Submitted to the Program in Computational Linguistics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

CARNEGIE MELLON UNIVERSITY

April 1993

Signature of Author........................................................
Program in Computational Linguistics
May 26, 1993

Certified by........................................................
Masaru Tomita
Associate Professor, Computer Science
Thesis Supervisor

Certified by........................................................
Jaime G. Carbonell
Professor, Computer Science
Thesis Supervisor

Certified by........................................................
David A. Evans
Associate Professor, Linguistics and Computer Science
Thesis Supervisor

Certified by........................................................
Alex H. Waibel
Senior Research Scientist, Computer Science
Thesis Supervisor

Certified by........................................................
Jun-ichi Tsujii
Professor, Centre for Computational Linguistics, UMIST
Thesis Supervisor

# Contents

# Preface

From theoretical linguistics to computational models of natural language, unification-based processing has become a central methodology in many research efforts. In theoretical linguistics, unification-based formalism has become one standard form of representation; many theories such as LFG ([Bresnan and Kaplan, 1982]), HPSG ([Pollard and Sag, 1987]), and JPSG ([Gunji, 1987]) use feature structure and unification as the base of constraint postulations. In computational linguistics, unification is used as the central constraint processing mechanism during parsing based upon the unification-based grammar analyses. In artificial intelligence, unification-based natural language is often used as an integral part of inference and learning mechanisms. Recent efforts in massively parallel artificial intelligence have also demonstrated the strength of graph unification as a uniform constraint processing mechanism for natural language in a massively parallel environment.

Despite the popularity of unification-based processing, graph unification, which is the computational method of unification-based processing, has remained a bottleneck of the unification-based systems. For example, in unification-based grammar parsing using parsing algorithms such as Earley's algorithm and Tomita's algorithm, unification operations often consume 85 to 95 percent of the total cpu time devoted to a parse. In one large-scale unification-based spoken language parser[1], sometimes 98 percent of the elapsed time is calculated to be devoted to unification operation alone ([Kogure, 1990]). Furthermore, the number of unification operations tends to grow as the grammar gets larger and more complicated. Thus, an unavoidable paradox is that when the natural language system gets larger and the coverage of linguistic phenomena is increased as an attempt to bring performance to a practical level, the number of unification

[1] ATR's HPSG-based spoken Japanese analysis system.

operations increases rapidly and the performance of the systems degrades to an impractical level. Thus an availability of efficient graph unification is of paramount importance both to theoretical natural language research as well as to practical natural language systems.

Overall parsing efficiency is crucial when building or experimenting with both practical and experimental natural language systems. For realtime practical systems, parsing speed is a prerequisite. For theoretical experimentation, the efficiency of hypothesis testing depends on the speed of constraint processing. In the modern linguistic framework, most parsing systems consist of two basic constraint processing mechanisms: 1) context-free parsing algorithms and 2) graph-unification algorithms. This thesis focuses on the efficiency gain of graph-unification algorithms. Given that most of the parsing time is consumed by graph-unification in large-scale NL systems, the benefit of improving graph-unification algorithms seems apparent.

The Center for Machine Translation of Carnegie Mellon University provided me with both the environment and the funding for pursuing my Ph.D. research at Carnegie Mellon University. The Laboratory for Computational Linguistics of Carnegie Mellon University was the base of the theoretical exploration of unification-based linguistic processing. Masaru Tomita was the chairman of my committee and provided me with everything I needed for pursuing my goals at Carnegie Mellon University, including the thesis topic. Jaime Carbonell, director of Center for Machine Translation and also one of my committee members, deserves many thanks for his support, advice and encouragement throughout my days at Carnegie Mellon University. Without his strong support, I would not have been able to continue my research at Carnegie Mellon. Also, without Tommy and Jaime, I would not have joined Carnegie Mellon University in the first place. David Evans, my advisor and also the director of the Laboratory for Computational Linguistics and of the Ph.D. program in Computational Linguistics, supported me throughout my graduate student years and also provided me with linguistic and philosophical insights into natural language. Alex Waibel, also my advisor, provided me with an excellent opportunity to work on spoken language input for the parsing systems. He is the head of the JANUS project which is the CMU side of the Japan/US/German trilateral video-interpreting-telephony project which uses the unification algorithm described in this thesis. He has also contributed greatly to my research in the neural net frameworks which is not covered in this thesis. Junichi Tsujii, professor at UMIST (University of Manchester Institute of Science and Technology, U.K.) and the only non-CMU member of my committee, has been providing me with new insights for natural language and machine translation for almost ten years. Carl Pollard, a former faculty member in the Computational Linguistics Program, my teacher of HPSG, and my advisor for my master's level thesis, originally introduced me to unification-based linguistics. Bob Carpenter and Rich Thomason were among other faculty members who gave me useful comments and support when I needed them. I wish also to thank Sergei Nirenburg, Lori Levin, Eric Nyberg, Teruko Mitamura, and Todd Kaufmann at the Center for Machine Translation for all the help and advice they gave me while I was at the Center. Radha Rao, the Business Manager of the Center, always handled my last minute requests with generosity, as did current and the past secretaries of the Center, namely, Cerise Josephs, Joan Maddama, and Barbara Moore. The former and the current members of the Laboratory for Computational Linguistics and the Department of Philosophy also contributed greatly to the research contained in this thesis. Among them are Ted Gibson, Alex Franz, Margalit Zabludowski, Sondra Ahlen, Marion Kee, and Renee Schafer. Important parts of the thesis research were conducted during the two periods when I was a Visiting Research Scientist at ATR (Advanced Telecommunication Research) Interpreting Telephony Research Laboratories in Kyoto, Japan twice (10 and 8 month stays in 1990 and 1991). Akira Kurematsu, Tsuyoshi Morimoto, Hitoshi Iida, Kiyoshi Kogure, Osamu Furuse, Susumu Kato, Masaaki Nagata, Toshiyuki Takezawa, Kenji Kita, Genichiro

Kikui, Toshihisa Tashiro, Kazumi Ohkura are among the researchers at ATR who contributed significantly to this work.

Makoto Takahashi, Hidehiko Matsuo, and Kyoko Sagi, of Toyo Information Systems worked with researchers at ATR and did their fully separate and independent implementations of my algorithms for ATR's large scale speech-to-speech translation systems (SLTrans and Asura). They have provided me with invaluable feedback for developing the later versions of my algorithms. Among their contributions were extremely detailed experimental results on the algorithms' behaviour under different environments (memory management, data structures, GC methods/timings, number of function calls, different parsers, different rule granularities, etc.) and comparisons with other unification algorithms which were conducted for almost two years using ATR's grammar (probably the largest Japanese grammar ever developed). Their test results confirmed my smaller scale experiments based on my implementations which are reported in this thesis. Marie Boyle of the University of Tuebingen, Peter Neuhaus of Universität Karlsruhe, and graduate students at Tokushima University are among other researchers who independently implemented the early versions of my algorithms and provided me with many useful and important suggestions. I am also indebted to the members of ICOT (Institute for New Generation Computer Technology) Natural Language Group for useful discussions, especially in the framework of unification and constraint-based natural language processing. Satoshi Tojo of Mitsubishi Research Institute, Hiroaki Kitano of NEC, Koiti Takeda of IBM, Akihiro Hirai, formerly with Hitachi, Hidefumi Sawai and Toru Matsuda of Ricoh, and Tsuyosi Kitani of NTT Data Communications Systems are among the members of the industrial affiliate program of the Center for Machine Translation who contributed greatly to the research contained in this thesis. Finally, special thanks to the following professors and researchers in Japan for their suggestions and help: Makoto Nagao, Hozumi Tanaka, Hidetoshi Shirai, Yuji Matsumoto,

Jun-ichi Nakamura, Koiti Hasida, Jun-ichi Aoe, Takako Fujioka, Keiichi Sudo, and Katashi Nagao. Mario Tokoro, Eiichi Osawa, and Katashi Nagao of Sony Computer Science Laboratories provided me with the environment for preparing the later versions of this thesis as well as an opportunity to test my algorithms on their multimodal systems. I originally came to the United States as a Fulbright Scholar in 1985. Thanks are due to the members of the Fulbright Committee and Senator Fulbright for giving me the opportunity to pursue my graduate research in the United States. (first at Yale and then at CMU). Roger Schank, Christopher Riesbeck, and Lawrence Birnbaum were among my first teachers of AI/NLP. Finally, David Rockefeller, Jr., Richard Vowell, Jotaro Takagi, Koyata Hosokawa, and Kiyoaki Hara are among the people outside of the natural language community who contributed greatly to my stay in the United States. Some parts of this thesis were written while I was visiting the Department of Environmental Information of Keio University and while I was a member of the faculty in the Department of Information Science and Intelligent Systems of Tokushima University. Kazuhiko Tsuda, Alfredo Maeda, Hideki Mima, Koiti Iriguchi, and Akiko Kita of Tokushima University and Lonnie Bartusis, Yukiko Uehara and Imari Karasawa of Carnegie Mellon University were extremely helpful in preparing the final version of this thesis.

# Chapter 1

# Foundations

## 1.1 Introduction

A variety of grammatical formalisms have been proposed historically in computational linguistics, natural language processing, and artificial intelligence to capture the phenomena called 'language'. Kay proposed Functional Grammar and Functional Unification Grammar (FUG, [Kay, 1984]) motivated by the notion of *functional description* of language. Bresnan and Kaplan developed the Lexical Functional Grammar (LFG, [Bresnan and Kaplan, 1982]) based on the framework of lexically-oriented linguistics. In the aritificial intelligence community, Definite Clause Grammar (DCG, [Pereira and Warren, 1980]) was developed by Pereira and Warren in the logic programming framework. Logic programming and DCG later became the base of natural language research efforts in the Japanese fifth generation computer research (ICOT). Gazdar developed Generalized Phrase Structure Grammar (GPSG, [Gazdar, *et al*, 1985]) in the nontransformational model of linguistic analysis. Pollard and Sag developed Head-driven Phrase Structure Grammar (HPSG, [Pollard and Sag, 1987]) in the similar nontransformational framework centered around the notion of *the linguistic head of a phrase.* Gunji developed the

Japanese Phrase Structure Grammar (JPSG, [Gunji, 1987]), which is a Japanese cousin of HPSG. JPSG later became the central linguistic processing framework in the Japanese interpreting telephony research efforts (ATR).

In the more computational and implementational aspects, PATR-II ([Shieber, *et al*, 1983]) was developed at the SRI AI Center as a theory-neutral *simple* and *mathematically well-founded* tool for natural language processing. At Carnegie Mellon University, to address the inefficiency of unification algorithms, pseudo unification and Pseudo Unification Grammar were developed as a part of machine translation research ([Tomita and Knight, 1987]).

All these grammatical formalisms (at least the modern versions of them) use feature structure as objects for capturing linguistic objects and use unification as the central constraint processing mechanism. In this chapter we would like to review both basic and formal properties of feature structures and unification.

## 1.2 Feature Structures

Despite the variety of analysis captured in modern theoretical and computational models of language, the so-called *feature structure* has been accepted as the common object for representation. Pollard and Sag explain, "Instead of the NASA Physicists' Euclidean spaces and differential equations, though, the formal object of choice in information-based linguistics are things known as *feature structures*". A feature structure is a structured object that represents informational content by specifying a set of *features* and their *values* pairs. Feature structures provide partial information about the information-bearing entities such as linguistic objects. In other words, feature structures are partial descriptions of things that are captured in different theories of language. Formally, feature structures can be understood as partial functions from *features* to their *values* where the underlying domain[1] of the partial functions is provided recursively. Conventionally, feature structures are represented using the matrices of feature value pairs. For example, a feature structure representing the linguistic object for a female professor named *Madoka* may be represented as below.

$$\begin{bmatrix} \textit{name} & \textit{Madoka} \\ \textit{sex} & \textit{female} \\ \textit{occupation} & \textit{professor} \end{bmatrix}$$

For the sake of economy of type-setting as well as of consistency with the sample feature structures in the appendix taken from the actual computer outputs, we also represent the same feature structure as below in this thesis.

```
[[name madoka]
 [sex female]
 [occupation professor]]
```

which can also be represented graphically as:

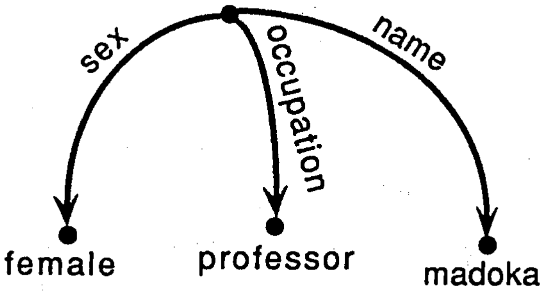


Figure 1-1: Graphical Representation of a Feature Structure

Since we will be representing feature structures in any of the above three ways in this thesis,

[1] I.e., As originally defined by Russell for domain of relations.

we will be using the terms ***features***, ***labels***, and ***labels on the arcs*** interchangeably.

A fundamental property of feature structures is their potential for ***hierarchiality*** ([Pollard and Sag, 1987]). Thus, the value of a feature itself may be another feature structure embedded within. For example, below is a feature structure providing a partial description of a linguistic entity for a third person singular feminine noun:

$$\begin{bmatrix} \textit{category} & N \\ \textit{agreement} & \begin{bmatrix} \textit{number} & \textit{singular} \\ \textit{person} & \textit{third} \\ \textit{gender} & \textit{feminine} \end{bmatrix} \end{bmatrix}$$

Or in our alternate notation:

```
[[category N]
 [agreement [[number singular]
             [person third]
             [gender feminine]]]]
```

Which can be graphically represented as Figure 1-2:

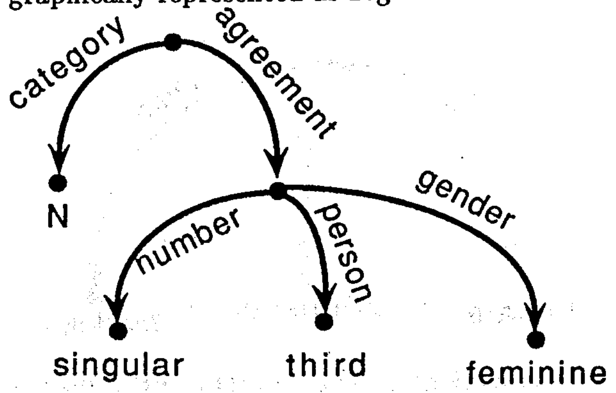


Figure 1-2: **A third person singular feminin noun entity**

Feature structures can be understood as partial functions mapping features to values. For example, in the example for *Madoka* provided above, the feature structure defines the mapping of the feature *name* to the value *madoka*, of *sex* to the value *female*, and of *occupation* to *professor*. The partial descriptions by feature structures can be understood as "descriptions participating in a relationship of partiality with respect to each other" ([Pereira and Shieber, 1984]). Formally, feature structures can be defined as below:

**Definition 1.2.1 (Feature Structures)** *Let $\mathcal{F}$ be a possibly infinite set of features and $C$ a possibly infinite set of atomic values. We first define special feature structures $\{\top, \bot\}$. $\top$ represents "no information" and $\bot$ represents "inconsistent information". The set $\Gamma$ of feature structures can be defined by:*

$$\Gamma = \bigcup_{i=0}^{\infty} \Gamma_i \cup \{\bot\}$$

*The set $\Gamma_i$ is defined recursively such that*

$\Gamma_0 = C \cup \{\top\}$

*For $i \geq 1$, $\Gamma_i = \{\gamma | \gamma : \mathcal{F} \xrightarrow{p} \bigcup_{k=0}^{i-1} \Gamma_k\}$ where $Dom(\gamma) \in \mathcal{P}'(\mathcal{F})$.*

*Here $Dom(\gamma)$ denotes domain of the partial function $\gamma$[2] and $\mathcal{P}'(\mathcal{F})$ denotes a non-empty power set[3] of $\mathcal{F}$ and $\xrightarrow{p}$ notates partial mappings.*

This way, the features $(l_1, ..., l_m)$ of a feature structure $\gamma$ are mapped to the values $(\gamma(l_1), ..., \gamma(l_m))$ for which the feature structure, as a partial function, is defined.

NOTATION. Following Pereira's notation ([Pereira, 1985]), we represent the value of the feature $l_i$ of a feature structure $\gamma$, i.e., $\gamma(l_i)$ by $\gamma/l_i$.

For example, $\gamma/category = N$ represents the feature structure provided above for the partial description of a third person singular feminine noun.

[2] Recall that in set theory, domain of a partial function f:A → B is the set $\{a | a \in A \text{ and } f(a) \in B\}$. Also image of f, written Im(f) is the set $\{f(a) | a \in Dom(f)\}$.

[3] Recall that power set of A is the set of all subsets of A. For example, if A={1,2,3}, power set of A is {1,2,3},{1,2},{2,3},{1,3},{1},{2},{3}, and ∅.

NOTATION. Following the standard notation of set theory, we shall write $Dom(\gamma)$ to denote the domain of the mappings from features to values for a feature structure $\gamma$. In other words $Dom(\gamma)$ represents the set of features on the feature structure $\gamma$. These features are not recursive, i.e., only the highest features are elements of $Dom(\gamma)$. For example, $Dom(\gamma)$ of the feature structure provided previously for the description of a third person singular feminine noun is $\{category, agreement\}$.

For embedded feature structures, we generalize the notion of features and introduce the notion of *path.* A path is a sequence of features from the outermost feature structure of the embedding to the feature of the innermost feature structure. In our vocabulary, it is a sequence of features from the highest to the lowest in the feature structure hierarchy. For instance, $< agreement, number >$ is a path for the sample feature structure provided above for a third person singular feminine noun.

NOTATION. We generalize the notation $\gamma/l$ representing the value of the feature l of feature structure $\gamma$ to apply to the path of features, written $\gamma/p$. Therefore, given the path $p \in \mathcal{F}^*$, which is $p =< l_1, ..., l_n >$ embedded in $\gamma \in \Gamma$, then $\gamma/p = (...((\gamma(l_1))(l_2))...)(l_n)$. For instance, with the above feature structure for a third person singular feminine noun, $\gamma/ < agreement, number >= singular$.

## 1.3 Four Types of Feature Structures

We have four types of feature structures: *atomic*, *complex*, *variables* and *inconsistency.* Atomic feature structures are feature structures with constant atomic values. Complex feature structures are feature structures which contain feature structures within. Thus, to be precise, only complex feature structures can be viewed as partial functions from features to values. The values of complex feature structures themselves are always feature structures (complex, atomic, or variable). Variables are the special feature structures with an empty domain. Variables are also called *Top* in this thesis[4]. Variables are the least informative feature structures indicating no information at all. *Inconsistency* are those that indicate inconsistent information. The idea behind *inconsistency* is that such feature structures represent more information than any feature structure possible. It indicates too much information to the extent that it is inconsistent. *Inconsistency* is also called *Bottom* in this thesis. Care needs to be taken in reading *Top* and *Bottom*, since due to historical reasons for looking at the hierarchy of information content, *Top* are somtimes called *Bottom* in the literature (and vice versa).

NOTATION. We shall denote *variables* by either [] or $\top$ and *inconsistency* by $\bot$.

## 1.4 Generalization and Unification

We can define two classical operations on feature structures: generalization and unification. Generalization is the operation to find a feature structure that contains only the information that is common in two feature structures. When common paths contain conflicting information, generalization introduces a variable and makes an abstraction. Unification is the operation to find a feature structure that contains the information in both feature structures but no additional information. If inconsistency of information is found at any depth of the paths, unification immediately returns *inconsistency* for the highest feature structure (the root feature structure). When a unification returns *inconsistency*, we say that the unification *failed*. This way, unification is a operation to determine the consistency of information between two feature structures. Informally, if $\gamma$ is a feature structure ( $\gamma \in \Gamma$ ), generalization of $\gamma$ and $\top$ always returns $\top$ since by definition no information is common between the two. Alternatively, unification of $\gamma$ and

[4] Note that [Tomabechi, 1991a] and [Tomabechi, 1992] called it *bottom* instead of *top*.

$\top$ always returns $\gamma$ since by definition, $\gamma$ contains the information of both and nothing more. Generalization of two atomic feature structures is $\top$ if they are not the same. That is, it has the property of making an abstraction by returning a variable for inconsistent information. If they are the same then the generalization is also the same. Unification of two atomic feature structures are $\perp$ if they are not the same. If they are the same then unification is also the same. Generalization of two complex feature structures is the feature structure only with paths that are common to two feature structures and unification of two complex feature structures is the one that contains both the unique paths and the common paths.

NOTATION. We shall denote generalization operation by $\coprod$ (or $\sqcup$ ) and unification operation by $\prod$ (or $\sqcap$) in this thesis.

More formally, generalization $\coprod$ and unification $\prod$ operations on feature structures $\Gamma$ are defined below: But first, it is useful to define the two operations *Complementarcs* and *Intersectarcs* between feature structures. These operations were originally provided in [Pereira, 1985] and are central to unification-based algorithms including the one we are proposing in this thesis.

**Definition 1.4.1 (Complementarcs and Intersectarcs)** *For two feature structures $\gamma_1, \gamma_2 \in \Gamma$, the following two operations are defined corresponding to the set-difference and set-intersection operations on domains of the partial functions.*

$Complementarcs(\gamma_1, \gamma_2) = \{(l, \gamma) \in \gamma_1 \mid l \notin Dom(\gamma_2)\}$

$Intersectarcs(\gamma_1, \gamma_2) = \{(l, \gamma) \in \gamma_1 \mid l \in Dom(\gamma_2)\}$

Notation. We shall denote $Complementarcs(\gamma_1, \gamma_2)$ by $\gamma_1 \ominus \gamma_2$ and $Intersectarcs(\gamma_1, \gamma_2)$ by $\gamma_1 \lhd \gamma_2$.

$Complementarcs(\gamma_1, \gamma_2)$ is the set of mappings of $\gamma_1$ from features to values with features that exist in $\gamma_1$ but not in $\gamma_2$. Since mappings are often represented by arcs, they are called

Complementarcs. *Intersectarcs*$(\gamma_1, \gamma_2)$ is the set of mappings of $\gamma_1$ from features to values with features that exist in both $\gamma_1$ and in $\gamma_2$. Note that values of Complementarcs and Intersectarcs are sets of mappings (i.e., partial functions) and not domains.

**Definition 1.4.2 (Generalization)** *Let $\Gamma$ be the set of feature structures as defined previously, then below is the definition of the generalization operation ( $\coprod$ ):*

*1)* $\forall \gamma \in \Gamma,\ \gamma \coprod \top\ = \top$

*2)* $\forall \gamma_1 \in \mathcal{C}\ \forall \gamma_2 \in \mathcal{C} \setminus \{\gamma_1\},\ \gamma_1 \coprod \gamma_2\ = \top$

*3)* $\forall \gamma \in \Gamma,\ \gamma \coprod \gamma\ = \gamma$

*4)* $\forall \gamma_1, \gamma_2 \in \Gamma \setminus \{\top, \bot,\} \setminus \mathcal{C},\ \forall l_1 \in Dom(\gamma_1 \lhd \gamma_2),\ (\gamma_1 \coprod \gamma_2)l_1\ = \gamma_1(l) \coprod \gamma_2(l)$

**Definition 1.4.3 (Unification)** *Let $\Gamma$ be the set of feature structures as defined previously, then below is the definition of the unification operation ( $\prod$ ):*

*1)* $\forall \gamma \in \Gamma,\ \gamma \prod \top\ = \gamma$

*2)* $\forall \gamma_1 \in \Gamma\ \forall \gamma_2 \in \mathcal{C} \setminus \{\gamma_1\},\ \gamma_1 \prod \gamma_2\ = \bot$

*3)* $\forall \gamma \in \Gamma,\ \gamma \prod \gamma\ = \gamma$

*4)* $\forall \gamma_1, \gamma_2 \in \Gamma \setminus \{\top, \bot,\} \setminus \mathcal{C},$

*if* $\exists l \in Dom(\gamma_1 \lhd \gamma_2),\ \gamma_1(l) \prod \gamma_2(l)\ = \bot$

*then* $\gamma_1 \prod \gamma_2\ = \bot$

*else* $\forall l_1 \in Dom(\gamma_1 \lhd \gamma_2), \forall l_2 \in Dom(\gamma_2 \ominus \gamma_1)$

$(\gamma_1 \prod \gamma_2)l_1\ = \gamma_1(l_1) \prod \gamma_2(l_1)$ *and* $(\gamma_1 \prod \gamma_2)l_2\ = \gamma_2(l_2)$

Below is the example of unification and generalization:

```
1.
[[category N]
 [agreement [[number singular]
```

```
             [person third]]]]

2.
[[category N]
 [agreement [[number singular]
             [gender feminine]]]]

3. Unification of 1,2:
[[category N]
 [agreement [[number singular]
             [person third]
             [gender feminine]]]]

4. Generalization of 1,2:
[[category N]
 [agreement [[number singular]]]]


5.
[[category N]
 [agreement [[number plural]
             [person third]]]]

6. Unification of 3,5
 Inconsistency

7. Generalization of 3,5
[[category N]
 [agreement [[number []]
             [person third]]]]
```

## 1.5 Some Formal Properties of Feature Structures

From the definition of generalization and unification, we can easily see that unification and generalization satisfy the usual formal laws of idempotency, commutativity, associativity and absorption. However, distributivity is not satisfied.

**Idempotent :**

$$\gamma_A \amalg \gamma_A = \gamma_A$$

$$\gamma_A \prod \gamma_A = \gamma_A$$

**Commutative :**

$$\gamma_A \amalg \gamma_B = \gamma_B \amalg \gamma_A$$

$$\gamma_A \prod \gamma_B = \gamma_B \prod \gamma_A$$

**Associative :**

$$(\gamma_A \amalg \gamma_B) \amalg \gamma_C = \gamma_A \amalg (\gamma_B \amalg \gamma_C)$$

$$(\gamma_A \prod \gamma_B) \prod \gamma_C = \gamma_A \prod (\gamma_B \prod \gamma_C)$$

**Absorptive :**

$$\gamma_A \amalg (\gamma_A \prod \gamma_B) = \gamma_A$$

$$\gamma_A \prod (\gamma_A \amalg \gamma_B) = \gamma_A$$

**not Distributive :**

$$\gamma_A \amalg (\gamma_B \prod \gamma_C) \simeq (\gamma_A \amalg \gamma_B) \prod (\gamma_A \amalg \gamma_C)$$

$$\gamma_A \prod (\gamma_B \amalg \gamma_C) \simeq (\gamma_A \prod \gamma_B) \amalg (\gamma_A \prod \gamma_C)$$

Unification and generalization are not distributive since the generalization introduces variables for inconsistent information and therefore, order of unification and generalization changes the results. For example, Figure 1-3 is the example where distributive law does not hold for unification and generalization.

Note that when [b n] and [b o] are generalized [b []] is returned. This ability to generalize inconsistent information makes unification and generalization not distributive to each other.

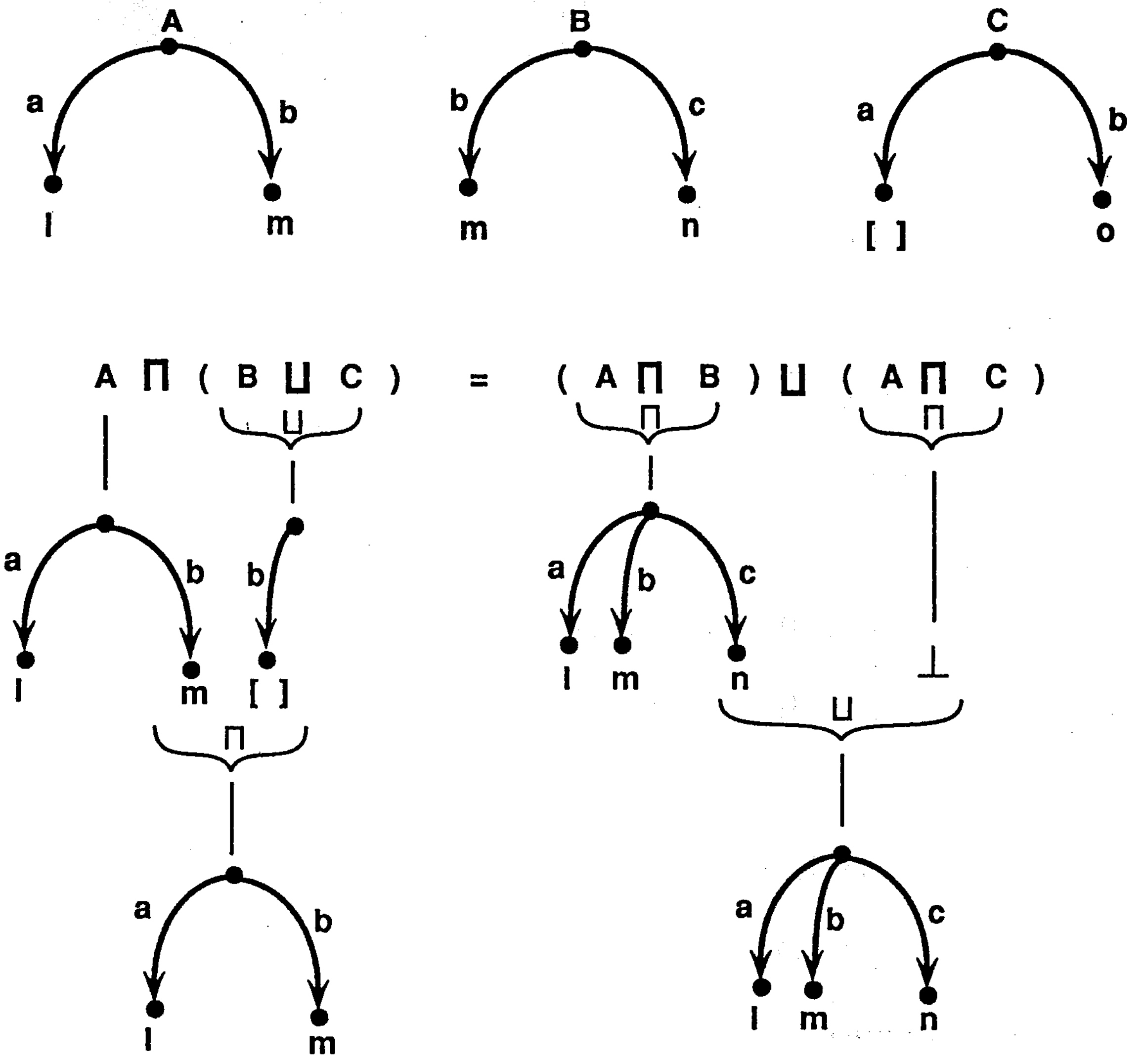


Figure 1-3: **Unification and generalization are not distributive**

However, since commutativity and associativity are satisfied, as long as unification is the only operation on feature structures, the order of unification does not matter regardless of the number of unifications performed on a set of feature structures. This is one important reason that feature structures and unification are used as formal tools for representing constraints in many linguistic theories, since constraints can be described declaratively without worrying about the order in which feature structures are combined by unification.

**lattice theoretic** When two operations such as $\vee$ and $\wedge$ are defined for some set $\mathcal{L}$ and if these two operations meet the laws of commutativity, associativity, and absorption, then we know in set theoretic terms that:

1. $a \vee b = b$ and $a \wedge b = a$ have the same values;
2. if we define as $a \leq b$ then $\mathcal{L}$ is a ordered set and forms a lattice.
3. Also $a \vee b$ and $a \wedge b$ are equivalent to join and meet operations on a lattice.

This way, from the properites we saw so far, we know that feature structures $\Gamma$ forms a lattice ($< \Gamma, \coprod, \prod >$) and that generalization is a join and that unification is a meet operations on lattice of feature structures. Here, the naturally defined order for feature structures corresponds to the order based upon how much information is contained in the feature structures. By definition of our generalization and unification operations, the maximum element of the lattice is $\top$ and the minimum is the $\bot$. In other words, more general elements are put toward higher parts of the lattice.

## 1.6 Reentrancy

An additional characteristic of feature structures that is found useful in modern theoretical and computational models of language is *reentrancy*. A reentrant feature structure contains another feature structure embedded within that is shared by two or more distinct paths within the feature structure.

NOTATION. We say that the values of two paths are "the same" when the two values are token identical. We use the notation $\equiv_\Gamma$ for this relation. Thus, $\gamma_x/p_1 \equiv_\Gamma \gamma_y/p_2$ when they are "the same", i.e., when $\gamma_x/p_1, \gamma_y/p_2$ are actually the single $\gamma_1 \in \Gamma$.

**Definition 1.6.1 (Reentrancy)** *Distinct paths $p_1, ..., p_m$ (all of them in a single $\gamma \in \Gamma$) are said to be reentrant iff $\gamma/p_1 \equiv_\Gamma \gamma/p_2 \equiv_\Gamma, ..., \equiv_\Gamma \gamma/p_m$.*

NOTATION. If paths $p_1, ..., p_m$ of a feature structure $\gamma$ are *reentrant* we shall denote the reentrancy by $[p_1, ..., p_m]_\gamma$

More informally, two or more distinct paths in the same feature structure are said to be *reentrant* when they share "the same" value. As an example, the feature structure below is not reentrant:

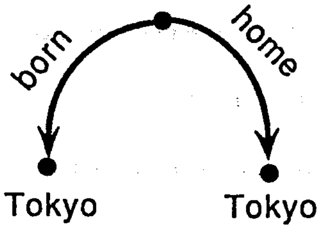


Figure 1-4: Non-reentrant feature structure

But the following is a reentrant feature structure:

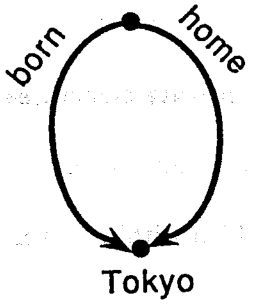


Figure 1-5: A reentrant feature structure

In our bracketed notation, the non-reentrant feature structure is:

```
[[born Tokyo]
 [home Tokyo]]
```

The non-reentrant feature structure above indicates the "similar value" but not the "same value". Thus, the above indicates the value *Tokyo* as the same type but not the same token. More precisely, the value may or may not be of the same token (i.e., the values may indeed be the "same", but they are only guaranteed to be "similar").

The reentrant feature structure in our bracketted notation is:

```
[[born X01 Tokyo]
 [home X01]]
```

Here, X01 shows that what follows it is reentrant with some other paths. If there is more reentrancy, more reentracy marks X01,...,Xmn will be tagged before the values. The tagging is valid only within one highest level feature structure ("highest" meaning the top level feature structure of the embedding). Therefore, the values with the same tagging X01 in the distinct feature structures do not indicate the same value.

We could also use the notation

$$\begin{bmatrix} \textit{born} & \boxed{1}\,\textit{Tokyo} \\ \textit{home} & \boxed{1} \end{bmatrix}$$

for representing the reentrancy.

Finally, below is an example of a reentrant feature structure with a complex value (embedding) using the three alternate notations.

As a graphic figure in Figure 1-6:

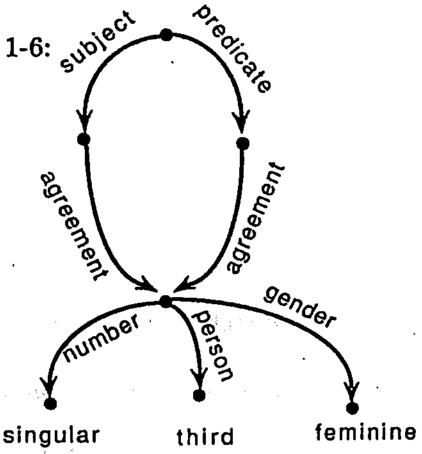


Figure 1-6: **A complex reentrant feature structure**

Using common notation found in linguistic literature:

$$\begin{bmatrix} \textit{subject} & \begin{bmatrix} \textit{agreement} & \boxed{1}\begin{bmatrix} \textit{number} & \textit{singular} \\ \textit{person} & \textit{third} \\ \textit{gender} & \textit{feminine} \end{bmatrix} \end{bmatrix} \\ \textit{predicate} & \begin{bmatrix} \textit{agreement} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

and in our bracketted notation.

```
[[subject [[agreement X01 [[number singular]]
                           [person third]
                           [gender feminine]]]]
 [predicate [[agreement X01]]]]
```

The ability to represent reentrancy has been useful and used standardizedly in many linguistic theories to capture phenomena such as *agreement* provided above. Also, by specifying the value of certain paths, the equality relation between the paths is an important tool for having different theoretical constraints interact. For example, an entity filling the position of a syntactic subject may be reentrant for an entity filling the semantic agent position of the feature structure.

## 1.7 Extention and Subsumption

"Some feature structures are *more informative* than others" ([Pollard and Sag, 1987]). As we saw previously, there is a natural partial order defined for feature structures forming a lattice. This ordering is based on the amount of informational content the feature structure carry. Informally, a feature structure $\gamma_A$ is considered more informative than $\gamma_B$ if it is at least as informative as $\gamma_B$ by containing at least all paths in $\gamma_B$. Such a relation is called *extention* and it is said that $\gamma_A$ extends $\gamma_B$.

NOTATION. If $\gamma_A$ extends $\gamma_B$ than we shall denote the relation by $\gamma_A \preceq \gamma_B$.

**Definition 1.7.1 (Extention Partial Order)** *The natural partial ordering $\preceq$ on $\Gamma$ for $<\Gamma, \amalg, \Pi>$ can be precisely defined as below*[5]*:*

*1.* $\forall \kappa \in \mathcal{C}$, $\bot \preceq \kappa \preceq \kappa \preceq \top$,

*2.* $\forall \gamma \in \Gamma \setminus \{\bot, \top\}$, $\bot \preceq \gamma \preceq \gamma \preceq \top$,

*3. if* $\gamma_A, \gamma_B \in \Gamma$ *are complex then* $\gamma_A \preceq \gamma_B$ *iff*

*(a) for each path* $< m_1, ..., m_i >$ *of* $\gamma_B$*, there is a path* $< l_1, ..., l_i >$ *of* $\gamma_A$ *such that* $\gamma_A / < l_1, ..., l_k > \preceq \gamma_B / < m_1, ..., m_k >$ *where* $1 < k \leq i$,

*(b) for every reentrancy* $[p_i, ..., p_k]_{\gamma B}$ *of* $\gamma_B$ *there is a corresponding reentrancy* $[p_j, ..., p_h]_{\gamma A}$ *in* $\gamma_A$.

The effect of the condition 3(b) above is that

[5] This definition is essentially the dual of Pollard's definition of $\sqsubseteq$ in [Pollard, 1984] except for the reentrancy handling part.

```
[[born X01 Tokyo]
 [home X01]]
```

extends ($\preceq$)

```
[[born Tokyo]
 [home Tokyo]]
```

This is so because the feature structure with the reentrancy is more informative than the one above without the reentrancy. In other words, a feature structure with "the same value" extends the feature structure with "the similar value". Because reentrancy is included in our definition of $\preceq$, the set $< \Gamma, \preceq >$ is not exactly the same as the lattice $< \Gamma, \coprod, \prod >$ which we discussed previously, that is so because our original formulation of unification operation provided at the beginning of the chapter did not take care of reentrancy. However, it is easy to see that $< \Gamma, \preceq >$ is a lattice, given that it is closed for $\top$ and $\perp$ (i.e, $\gamma_x \vee \gamma_y$ and $\gamma_x \wedge \gamma_y$ exist for any $\gamma_x, \gamma_y \in \Gamma$ where $\vee$ denotes *join* (least upper bound) and $\wedge$ denotes *meet* (greatest lower bound) . We will also provide a framework for unification operation that handles reentrancy in the later parts of this thesis. Then $< \Gamma, \coprod, \prod >$ will be the same lattice as $< \Gamma, \preceq >$.

Our definition of extention makes it clear that $\preceq$ is a partial order. This is so because we can see from our definition that for $\gamma_A, \gamma_B \in \Gamma$, it follows that (1) $\gamma_A \preceq \gamma_A$; (2) $\gamma_A \preceq \gamma_B$ and $\gamma_B \preceq \gamma_A \Rightarrow \gamma_A = \gamma_B$; (3) $\gamma_A \preceq \gamma_B$ and $\gamma_B \preceq \gamma_C \Rightarrow \gamma_A \preceq \gamma_C$. It is not a complete order, however, since not every feature structure in $\Gamma$ is in an extention relation.

The *dual* of the extention relation is the subsumption relation. $\gamma_A$ subsumes $\gamma_B$ if $\gamma_A$ is less informative than $\gamma_B$.

NOTATION. If $\gamma_A$ subsumes $\gamma_B$ than we shall denote the relation by $\gamma_A \sqsubseteq \gamma_B$.

**Definition 1.7.2 (Subsumption Partial Order)** *The partial ordering $\sqsubseteq$ on $\Gamma$ can be defined as the dual of $\preceq$, that is, for $\gamma_A, \gamma_B \in \Gamma$, $\gamma_A \sqsubseteq \gamma_B$ iff $\gamma_B \preceq \gamma_A$.*

Figure 1-7 is a conceptual diagram of a lattice of feature structures. Note that [] is put at the top of the lattice. Therefore, the higher the location of the lattice, the less informative (or more general) the feature structure becomes.

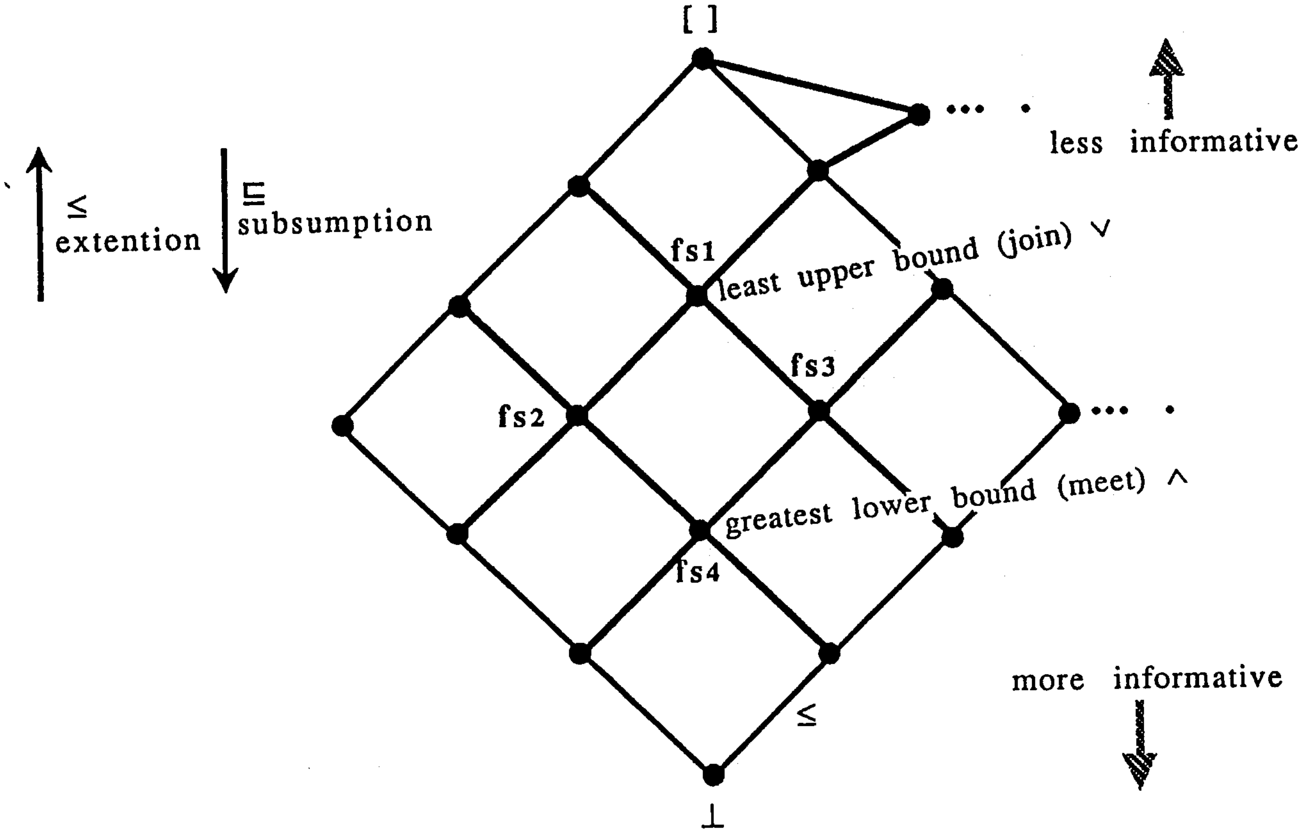


Figure 1-7: Lattice of Feature Structures

Below are the examples of the subsumption relation taken from [Shieber, 1986] (but using our notational convention). The feature structures provided earlier subsume all the feature structures provided later in these examples. In other words, the subsumption relation 1 ⊑ 2 ⊑ 3 ⊑ 4 ⊑ 5 ⊑ 6 holds.

```
1.
[]

2.
[[category N]]
```

```
3.
[[category N]
 [agreement [[number singular]]]]

4.
[[category N]
 [agreement [[number singular]
             [person third]]]]

5
[[category N]
 [agreement [[number singular]]
             [person third]]
 [subject [[number singular]
           [person third]]]]

6.
[[category N]
 [agreement X01 [[number singular]]
                 [person third]]
 [subject X01]]
```

## 1.8 Unification and Generalization Revisited

Unification is the least informative feature structure which contains all the information from both feature structures (but no additional information). By using the notion of extention partial order, we can say that **unification is the least informative feature structure that extends the two feature structures.** (That is, it is the *greatest lower bound* of two feature structures with respect to the extention ordering $\preceq$.) Thus, unification of two feature structures $\gamma_A$ and $\gamma_B$ is the least informative feature structure $\gamma_C$ that $\gamma_C \preceq \gamma_A$ and $\gamma_C \preceq \gamma_B$. Since extention and subsumption are duals, unification can also be defined as the most informative feature structure which is subsumed by two feature structures. That is, the unification operation returns the most informative feature structure $\gamma_C$ such that $\gamma_A \sqsubseteq \gamma_C$ and $\gamma_B \sqsubseteq \gamma_C$.

Similarly, generalization can be defined using the notion of extention partial ordering. **Generalization is the most informative feature structure that subsumes the two feature structures.** That is, $\gamma_C = \gamma_A \sqcap \gamma_B$ if $\gamma_C$ is the most informative feature structure with which $\gamma_C \sqsubseteq \gamma_A$ and $\gamma_C \sqsubseteq \gamma_B$ holds.

Note that by our definition of the subsumption relation, a reentrant feature structure extends a non-reentrant counterpart with similar values. Therefore, the unification of $\gamma_A$ and $\gamma_B$ in Figure 1-8 is $\gamma_B$.

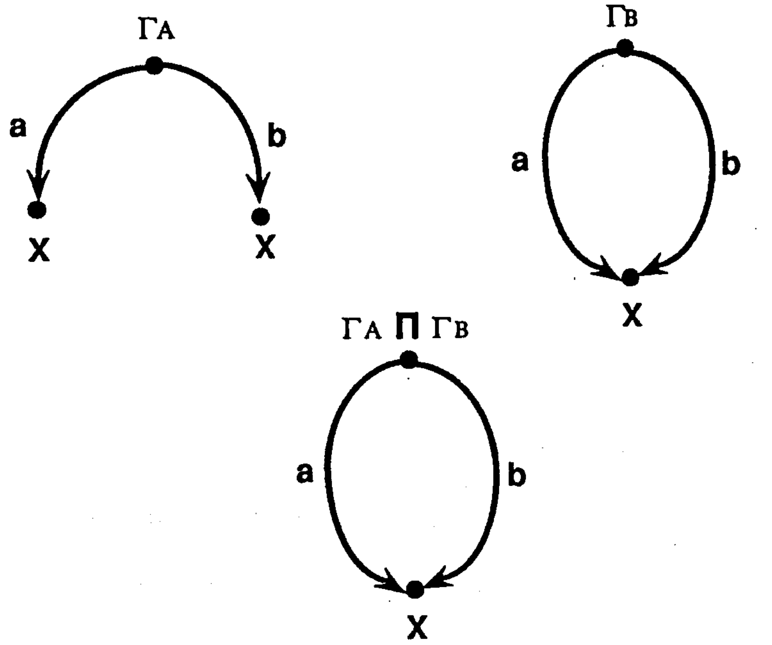


Figure 1-8: **Unification of same values with similar values**

The important properties of the operations on feature structures are that unification always adds information and generalization always subtracts information (unless two feature structures are already in the subsumption relation). Because of the monotonic information combining nature of unification, unification has been chosen as the central and often only operation on feature structures in many theoretical and computational models of language. The linguistic theories that use unification as the central (or only) method of capturing linguistic constraints are collectively known as "unification-based" theories. GPSG, LFG, FUG and HPSG are some of the well-known examples of unification-based theories. Below is the summary of the properties of unification which form the basis of formal representation of linguistic constraints in unification-based theories.

**Monotonicity:** Unification always adds information and never subtracts information. By definition, the result of unification is always subsumed by the two input feature structures. This property of unification is the reason that unification is used as the basic tool of combining information in many linguistic theories.

**Order Independency:** Since the unification operation meets the laws of idempotency, commutativity, and associativity, the order of unification operations is unimportant. This makes the feature structure and unification operation useful tools for capturing linguistic constraints declaratively.

**Undecidability:** The unification of two feature structures may be inconsistent ($\perp$). Two feature structures may contain incompatible information (incompatible paths). When the unification of two feature structures is inconsistent, it is conventional to say that the unification *fails*. We also say that the unification is *undecided* when it is $\perp$. Because the unification may fail, the unification operation can be used as an apparatus to check constraint satisfaction as well as to combine and propagate constraints.

We have seen that the usual formal laws of idempotency, commutativity, associativity, and absorption hold on unification forming a lattice. However, one final note here is that as we have seen before, feature structures are not distributive on unification and generalization. Therefore, if the generalization operation is adopted as a part of a constraint checking mechanism, the order of the applying generalization and unification would become important. This is part of the reason that generalization is not commonly used in computational models of language. [6]

[6] Disjunctive operations however, would be distributive with unification forming a ditributive lattice with unfication. For example, Pollard and Sag say about the distributive nature of unification and disjunction of feature structures that the extention ordered lattice of feature structures form a distributive lattice, i.e., $\gamma_1 \vee (\gamma_2 \sqcup \gamma_3) = (\gamma_1 \vee \gamma_2) \sqcup (\gamma_1 \vee \gamma_3)$ and $\gamma_1 \sqcup (\gamma_2 \vee \gamma_3) = (\gamma_1 \sqcup \gamma_2) \vee (\gamma_1 \sqcup \gamma_3)$. Thus many systems that use feature structures use disjunctive feature structures. However, the feature structures in this thesis are not disjunctive. That is because non-disjunctive unification algorithms can be extended into disjunctive ones by either 1) modifying the algorithm itself, 2) opening the disjunctive feature structures into cross multiples, 3) treating the disjunctive part and non-disjunctive part separately. The third method was developed by [Kasper, 1987] and our experiments show that it is the best method for a large scale grammar. We have seen that the major portion of the unification operation during a parse of a large-scale grammar is occupied by processing the non-disjunctive part (normally more than 95 percent) while very little is occupied by processing the disjunctive part (less than 4 percent or so). Thus, it makes sense to adopt Kasper's method and process the disjunctive part separately instead of making unification algorithms heavy by introducing the capability to processing disjunctive feature structures.

# Chapter 2

# Graph Unification in Natural Language

## 2.1 Feature Structure Graphs

Feature structures were chosen as the formal objects for representation of linguistic entities in modern theoretical and computational linguistic theories such as Functional Unification Grammar (FUG [Kay, 1984]), Lexical Functional Grammar (LFG [Bresnan and Kaplan, 1982]), Generalized Phrase Structure Grammar (GPSG [Gazdar, *et al*, 1985]) and Head-driven Phrase Structure Grammar (HPSG [Pollard and Sag, 1987]) and are commonly known as *feature structures* which are feature-theoretic structures of feature/value pairs. For example, an HPSG-like lexical entry for the word *laughs* may look like this:

```
[[PHON laughs]
 [CAT [[HEAD [[MAJ verb]
              [VFORM finite]
              [AUX minus]
              [INV minus]
```

```
                [PRD minus]]]
         [SUBCAT [[FIRST [[CAT [[HEAD [[MAJ noun]
                                       [NFORM nom]
                                       [PERS third]
                                       [NUM sing]]]]]
                             [CONT [[ARG1 X01]]]]]
                    [REST end]]]]]
 [CONT [[RELATION laugh]
         [ARG1 X01]]]]
```

By representing mappings captured by feature structures as arcs on nodes, directed graphs are commonly used for both graphic and computational representation of feature structures. In directed graph representations, *features* are represented as labels on the directed arcs and *values* are represented as nodes. Generally the labeled directed graphs used to represent linguistic *feature structures* have the following properties:

- **Arcs represent features:** Each feature in a *feature structure* is explicitly represented by a corresponding arc in the graph representation of *feature structures*.
- **Arcs are labelled:** Arcs are labeled to represent feature labels.
- **Arcs are unordered:** The order of arcs in the same level of a feature structure graph is irrelevant to the expressed content of the feature structure. Thus, the order of arcs contained in the nodes has no significance. (That is, arc lists in the nodes are actually *sets*, not *lists*) .
- **Arcs are directed:** *Feature structures* are partial functions mapping features to values. Since this mapping is unidirectional, the arcs representing the mappings are directed.
- **The number of arcs is not fixed:** The number of mappings from features to values captured by a feature structure can be finitely many. There is no fixed limit on the number of arcs in a feature structure.

- **Nodes represent feature values:** A node represents either: 1) an atomic value 2) a complex value or 3) a variable. Variables need to be represented specifically as nodes (feature structure) since a variable may be shared by multiple paths (reentrancy).

- **Graphs may contain convergence:** A feature structure may be reentrant; therefore a feature structure graph may contain a convergence.

- **Graphs may contain cycles:** A path in a feature structure may be cyclic either because grammar allows for cyclicity or because the unification of two reentrant feature structures created a cycle.

The last point about cyclicity requires some explanation. Our definition of feature structure in the previous chapter did not include the nature of cyclicity. In fact, most unification-based theories assume feature structures to be acyclic. However, some grammar formalisms allow for cyclicity in constraint graphs. The ATR grammar that we used for experiments for this thesis is one of them. Also, it is often easier to represent some linguistic phenomena using cyclic feature structures such as constraints on relative clauses even with grammatical formalisms that assume no cyclicity. Finally, unification of two reentrant feature structures may result in a cyclic feature structure even if the input grammar specifications did not have cyclic paths at all. Thus, it is safe to assume that feature structures may be cyclic even if the grammatical formalism did not assume cyclicity. Therefore, feature structures need to be represented as directed graphs (dgs) and not as directed acyclic graphs (dags) if we would like to design a unification-based systems with robust behaviour. In fact, our definition of extention and subsumption in the previous chapter already included the possibility of cyclic feature structures. Hereafter in this thesis, we assume feature structures to be directed graphs and not acyclic directed graphs.

Below is an example of grammatical rule entries taken from ATR's grammar [Takahashi, *et al*, 1992]. The first rule formalizes the subcategorization principle and the second rule represents the adjunct (or COH) principle in JPSG. Although neither of the rules are cyclic (only reentrant), a cyclic feature structure will result when these rules are combined (unified). Given that these are very frequently used rules, it is important that natural language systems using a grammar like this one accept cyclic feature structures and handle them efficiently.

```
1) <syn subcat> = <dtrs 2 syn subcat rest>
   <dtrs 1 syn head coh> = <dtrs 2 syn sub cat first>

2) <dtrs 1 syn head coh> = <dtrs 2>
```

These path equations indicate the convention that paths equated by = point to the same node (variable). That is, $\gamma_0 / < syn\ subcat >= \gamma_0 / < dtrs\ 2\ syn\ subcat\ rest >$, etc. The resulting graph of the unification of the rules looks as the following (taken from [Takahashi, *et al*, 1992]):

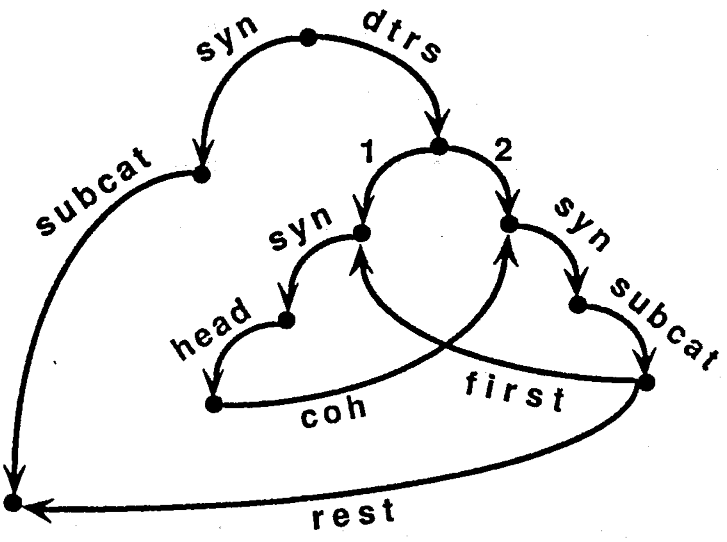


Figure 2-1: Grammatical rule with a cycle

## 2.2 The Nature of Graph Unification

While the unification operation has been popularly adopted as a basic tool in theoretical and computational models of language, the design of an efficient unification mechanisms has not been an easy task. Normally, unification is by far the most computationally expensive part of natural language systems. For example, considering the time-efficiency problem alone, in typical large-scale systems such as [Morimoto, *et al*, 1990], 75 to 95 percent of parsing time is occupied by unification alone. Also, designing an efficient unification operation that meets the properties of *feature structures* listed in the previous section is not an easy task. Recall that these properties include 1) order independence, 2) unfixed number of arcs, 3) convergence, and 4) cyclicity. Below are some of the essential criteria that a graph unification method for natural language processing must meet:

- **The input graphs should not be destroyed:** The input graphs must be preserved because constraints are represented by feature structures as rules that are unified against the feature structures that are produced by input. Since rules are used many times, the original graph representing the rule needs to be preserved. Also, during the analysis of the input language, constituent graphs representing the current hypothesis of the constituency are created. Since these constituent graphs are also applied many times against different hypotheses, these graphs need to be preserved as well. Consequently, in most unification algorithms, nodes are copied prior to or during unification causing a heavy overhead on unification operations.

- **Graphs may contain convergence and cycles:** As discussed in the previous sections, feature structures may be reentrant and even cyclic. As we have seen, the possibility of reentrancy complicates the nature of the subsumption relation as well as introduces the bidirectionality of the information flow, since different parts of a feature structure may be connected by reentrant paths and since whatever happens in one reentrant path must also be reflected in other reentrant paths. The difficulty of handling cyclicity is even more problematic, since 1) cyclicity also changes the nature of the subsumption relation and consequently the nature of unification; and 2) cyclicity may cause an infinite loop during a unification. With respect to the problem of a loop: If we perform a vanilla 'occur check' to avoid the loop, this check would require a scan through the entire graph for one extra pass; this can be very expensive if a graph is large. Therefore, an efficient method for graph unification must have a built-in and cheap mechanism for handling cycles.

- **Graphs may contain variables:** Some feature structures are variables. Often variables are introduced to capture the reentrant constraints on the equality of path destinations. Such a construct is frequently used in grammatical specifications of phenomena such as *agreement.* As we have seen in the previous chapter, variables ([],⊤) have peculiar behaviour for subsumption (and therefore for unification). Again, correct and cheap handling of such a behaviour is a basic requirement for an efficient graph unifiction method.

The difficulty of handling reentrancy, cycles, and variables becomes even more problematic when they are combined. Below is an example taken from [Pollard and Sag, 1987] (with a different notation).

```
dg1
[[a [[a X01]]]
 [b X01]]

dg2
[[a X02]
 [b [[a X02]]]]
```

```
dg3
[[a X01 [[a X02]]]
 [b X02 [[a X01]]]]
```

The unification of dg1 and dg2 will result in dg3. However, Pollard and Sag would not count dg3 as a valid feature structure[1] because they define subsumption differently, and cyclicity is not allowed in the graphs. Therefore, this unification would be counted as a 'fail' by them.

Let us examine the above three feature structures graphically. Figure 2-2 is the directed graph representation of the three feature structures:

As we can see from the figures, clearly, dg3 extends both dg1 and dg2, using our definition of extention/subsumption. Viewing dg3 as the least informative feature structure that is subsumed by both dg1 and dg2 seems perfectly reasonable. In fact, in the current frameworks of unification-based processing, our definition of subsumption is often adopted and has been proven useful in many systems (including CMU, ATR, and Tokushima systems). Whether cyclicity in the graphs is originally assumed or whether it is an avoidable result of allowing for reentrancy and of providing a definition of subsumption that covers reentrancy, the above unification result must naturally be accepted as a unification success. Otherwise, reentrancy is not fully processed in unification-based systems. In other words, once a unification-based framework adopts reentrancy and variables, it has no choice but to adopt cyclic feature structures in order to handle reentrancy and variables adequately.

It should be easy to imagine that the design of a unification methodology that covers the unification of dg1 and dg2 to result in dg3 is not trivial. Once cyclicity is allowed for feature structures, we can have a multiple loops in different places of a feature structure. Thus, it is important that an efficient unification method provide natural and cheap functionalities for

[1] The recent versions of the HPSG theory, however, also treat dg3 as the valid unification of dg1 and dg2 (according to Bob Carpenter, personal communication).

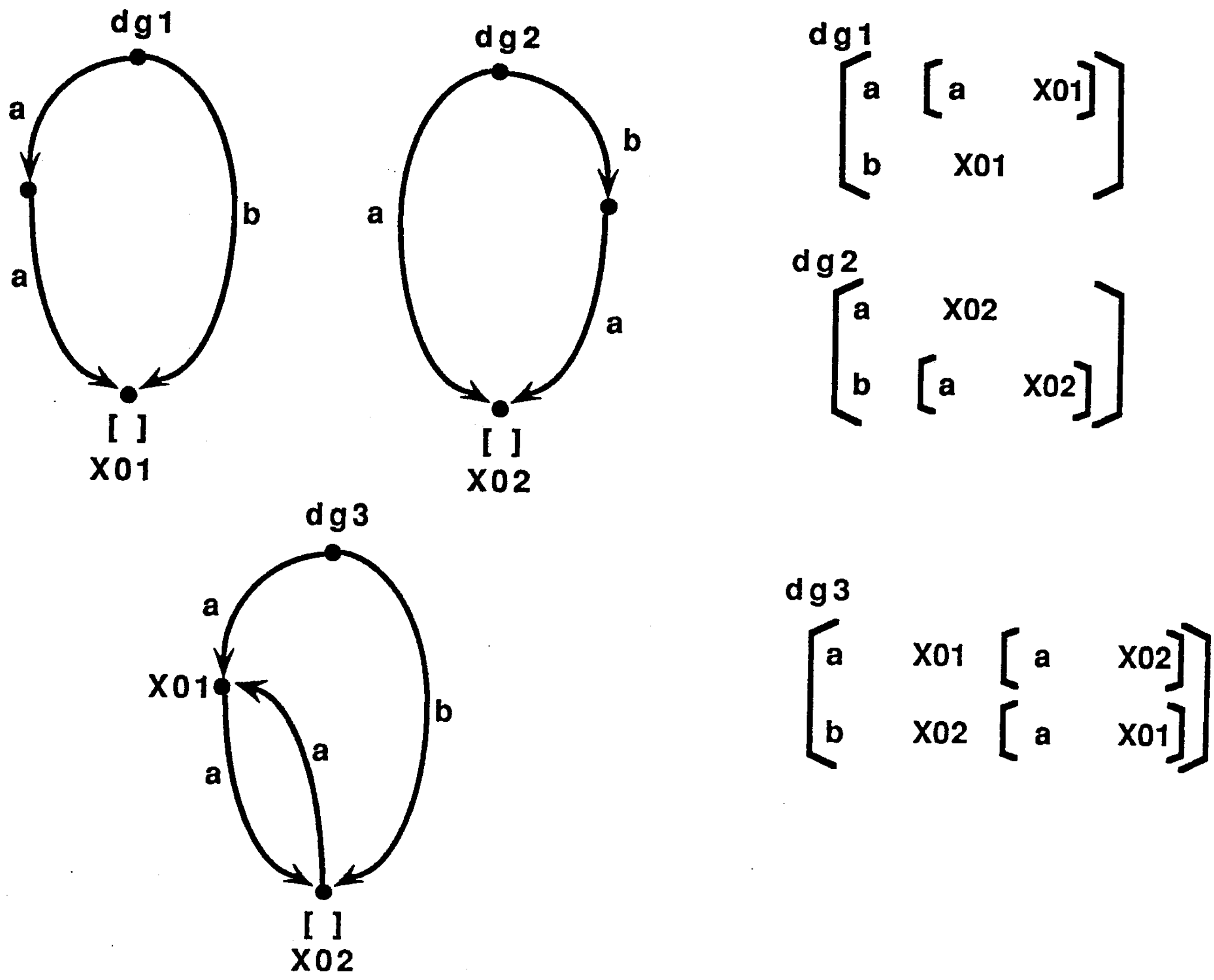


Figure 2-2: **Unification resulting in a cycle**

handling cycles adequately. We will see in Chapter 4 that one of the main advantages of the algorithm described in this thesis is the ablility to handle cycles naturally.

## 2.3 Unification and Parsing

Before discussing the actual unification algorithms, we would like to briefly review a representative methology in unification-based natural language processing. At least three methods are common in using unification during natural language processing. The first method is employed when linguistic theories such as HPSG are directly implemented. Lexically-oriented theories such as HPSG assume no separate context-free rule for phrase structures. Phrase structure rules are implicitly contained in subcategorization lists which are lexically stored. Therefore, combined with universal principles such as the Head-Feature Principle ([Pollard and Sag, 1987]), which are also represented through *feature structures*, parsing is performed purely through graph unification [Franz, 1990]. The second method which is most popular ([Shieber, *et al*, 1983],[Tomita and Carbonell, 1987],[Morimoto, *et al*, 1990]) is employed when grammatical theories such as LFG and GPSG, which assume context free rules, are adopted. Also, some systems (such as [Morimoto, *et al*, 1990]) use this method for speed, although they adopt lexically-oriented formalisms (such as HPSG) by extracting lexically-specified subcategorization constraints as context-free rules. In these systems, context-free rules based upon major grammatical categories (parts of speech) are augmented with unification-based constraints that specify actual constraints for building up phrase structures. The third method is employed when graph-based constraints are used in the conceptual memory-based recognition of natural language inputs. In these systems (which often assume massively-parallel spreading activation architectures) graphs are propagated in the network of semantic memory nodes to provide syntactic constraint application while performing spreading activation-based conceptual memory recognition. Here, we would like to examine the second method, which is the method adopted in the majority of natural language systems. We will not discuss the first method in this thesis. Because the first method has no separate control structure other than the unification operation itself. We will not discuss the third method in this thesis either. (Please refer to [Tomabechi and Levin, 1989], [Tomabechi, 1991b], [Tomabechi, 1991c] for discussions of the third method.)

In the so-called augmented context-free parsers, grammar rules are provided to postulate how major grammatical categories (phrase types) combine to create larger phrasal structures through context free grammars (CFGs). But whereas a context-free grammar allows only a finite number of predefined atomic phrase types or *nonterminals*, a unification-based (augmented) context-free grammar implicitely defines an infinity of phrase types ([Pereira, 1985]). A phrase type is specified for type X0 by postulating the context free rule X0 ⇒ X1, ... ,Xn (where X1, ... , Xn represent its constituents), which is augmented by equations specifying values for x0/P1,...,X0/Pn. The values for X0/P1, ..., X0/Pn may be *atomic* as well as *complex* specified by Xm/Pm. Thus a rule entry may look as Figure 2-3.

```
X0 => X1,...,Xn
X0/P1 = a1
X0/P2 = X1/P1
X0/Pi = Xm/Pm
      :
      :
```

Figure 2-3: **An augmented CFG rule entry**

Here a1 is an atomic value and Xn/Pn represents the node at the end of Xn through the path Pn. A sample rule entry using the commonly adopted PATR-II ([Shieber, *et al*, 1983]) notation for augmented context free rules looks follows:

```
X0 => X1 X2
<x0 cat> = VP
<x1 cat> = V
<x2 cat> = N
<x0 head> = <x1 head>
<x2 head case> = objective
<x1 head vtype> = transitive
<x0 cont> = <x1 cont>
```

Figure 2-4: **An augmented CFG rule using PATR-II notation**

Here ⟨ Xn .... ⟩ specifies the paths.

Most implementations omit ⟨ Xn cat ⟩ = Cat and specify the major categories directly in the context-free parts (e.g. VP ⇒ V N) so that parsing algorithms ([Earley, 1968], [Tomita, 1985], etc.) can be directly used on the context-free portion of the rule entries. Therefore, rules in most systems look like Figure 2-5:

```
VP => V N
<x0 head> = <x1 head>
<x2 head case> = objective
<x1 head vtype> = transitive
<x0 cont> = <x1 cont>
```

Figure 2-5: **a standard ACFG notation**

The augmentation path equations are converted to graphs when the grammar is read into the system. These graphs are stored along with the context-free rules. For example, the graph provided in Figure 2-6 representing the path equation above is stored with the rule VP ⇒ V N.

Furthermore, lexical entries (i.e., terminal symbols) are augmented with path equations. Augmentations are also converted into graphs when the grammar is read into the system. Figure 2-7 is a sample lexical entry for the verb *laughs* based on the HPSG framework.

Whenever a context free parser fires the rule VP ⇒ V N, (the subgraph of x0 of) the constituent graph, which was stored along with the lexical entry (such as *laughs*) that fired V, is unified with (the subgraph of) x1 in the augmentation, and the constituent graph for N is unified with (the subgraph of) x2 in the augmentation. If unification fails, then the rule is killed. If unification succeeds then (the subgraph of) x0 is the result of the rule application. If VP is used subsequently (for example by a rule S ⇒ VP), then the x0 path of the result graph will be unified with the rule augmentation graph.

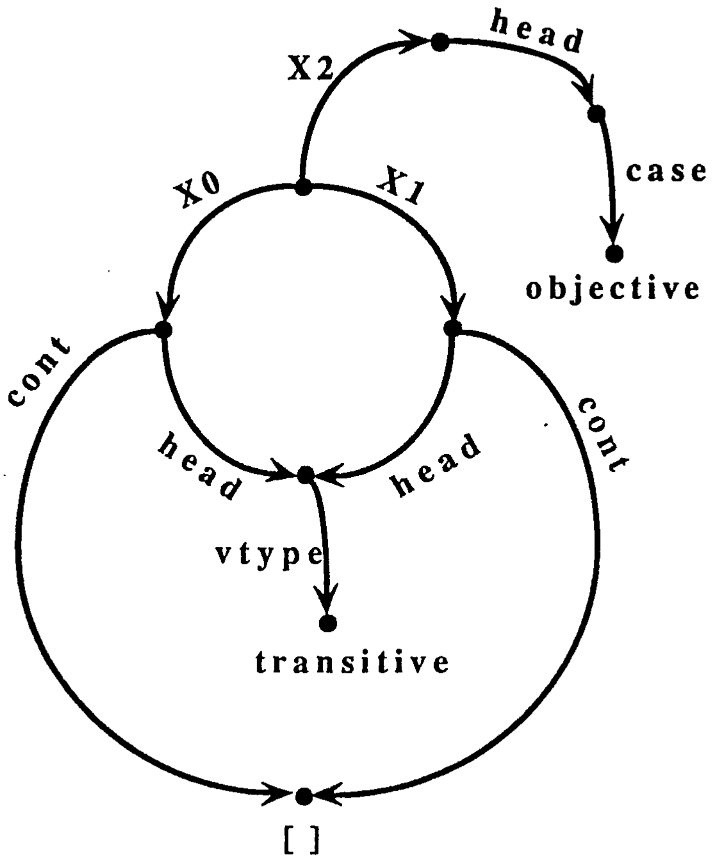


Figure 2-6: **An augmentation graph for VP ⇒ V N**

This way, parsing of the augmented context free grammar continues by repeatedly firing rules and unifying constituent graphs with the rule augmentation graphs. Graph unification performs the functions of 1) applying constraints (blocking unacceptable constituents and inapplicable rules) 2) building the larger information contents by unifying two graphs and 3) propagating information upward in the constituency (the bar levels [Jackendoff, 1977]). Everytime a rule is fired, graph unifications between a rule and a constituent occurs for each Xn and therefore, the number of unifications performed during a parse can be massive.

```
V -> <l a u g h s>
<x0 head vform> = finite
<x0 head aux> = minus
<x0 head inv> = minus
<x0 head prd> = minus
<x0 subcat first cat head nform> = nom
<x0 subcat first cat head pers> = third
<x0 subcat first cat head num> = sing
<x0 subcat first cont arg1> = <x0 cont arg1>
<x0 subcat rest> = end
<x0 cont relation> = laugh
```

**Figure 2-7: a lexical entry for *laughs***

# Chapter 3

# Past Representative Methods

## 3.1 Pereira's method

Pereira ([Pereira, 1985]) proposed a method of directed acyclic graph (dag) unification based upon the notion of structure-sharing. The basic idea behind his scheme is that an original dag and the result dag can share the information (structure) except for the information that modified the original dag as a result of unification. Therefore, a result graph is represented as a combination of the original graph and the information that represents the changes that are caused by unification.

```
+---------------+
|   skeleton    |  <== Pointer to the original dag structure
+---------------+---------------+
|               | rerouting     | <== forwarding pointer
+  environment  +---------------+
|               | arc-binding   | <== new arcs to be added to create result
+---------------+---------------+
```

Figure 3-1: **Pereira's Data Structure**

In this scheme, a dag is represented by a *skeleton* and an *environment*. *Skeleton* represents the (pointer to the) original dag. *Environment* contains the information that represents the changes to be made in order to create a result dag.

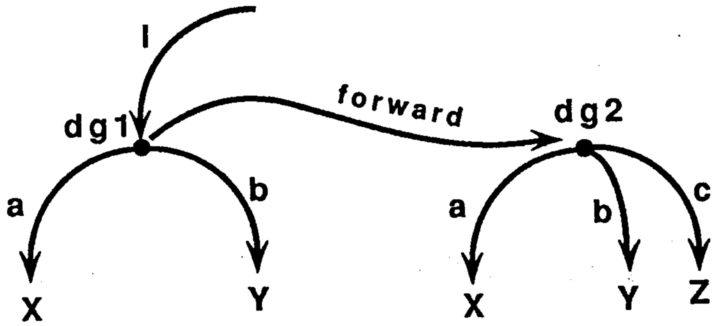


Figure 3-2: **Forwarding Operation**

Specifically *environment* contains *rerouting* and *arc binding*, which represent the forwarding information and new arcs to be added to create the result, respectively. Forwarding redirects a reference of a particular graph node to some other node. In Figure 3-2, the reference to the top node of the left graph is forwarded to the right graph; therefore externally the content of the left graph looks like the content of the right graph. This operation of putting the forwarding pointer on a node is called a *forwarding* operation; following the pointer to return the intended content is called *dereferencing.* In Pereira's method, when two graphs (dagl and dag2) are unified and if recursions into shared arcs succeed, then dagl's highest (root) node is forwarded to dag2's highest node (as in the figure). This forwarding pointer is saved into the *rerouting* content of the *environment.* Also, the complementarcs[1] in the highest node in dagl and the highest node in dag2 (i.e., the arcs with labels that exist in dagl but not in dag2) are added to the *arc-binding* field of *environment* to represent necessary changes for producing the result

[1] **As we defined in Chapter 1.**

graph based upon dag2's *skeleton.* This adding to the *binding* content of the *environment* is performed at all depths of recursion so that a result graph can be created (when necessary) by looking at the dag2 nodes and referencing the environment. This way, no copies at all are made in his method. Instead the result graphs are dynamically created when the graph is needed later. Below is Pereira's algorithm taken from [Pereira, 1985] (slightly modified to make it up-to-date):

**PEREIRA'S ALGORITHM**

```
FUNCTION unify (dag1-underef,dag2-underef);
  dag1 ← dereference(dag1-underef);
  dag2 ← dereference(dag2-underef);
  IF (dag1 $\equiv_\Gamma$ dag2) THEN
     return(dag2);
  ELSE IF (dag1 $\equiv_\Gamma$ Top) THEN
     forward(dag1,dag2);[2]
     return(dag2);
  ELSE IF (dag2 $\equiv_\Gamma$ Top) THEN
     forward(dag1,dag2);
     return(dag1);
  ELSE IF (dag1 and dag2 are atomic and the values are equal) THEN
     forward(dag1,dag2);
     return(dag1);
  ELSE IF (dag1 and dag2 are complex) THEN
     shared ← intersectarcs(dag1,dag2);
     new ← complementarcs(dag1,dag2);
     forward(dag1,dag2);
       FOR EACH arc IN shared DO
              unify (destination of
                           the shared arc for dag1,
                     destination of
                           the shared arc for dag2);
       If all recursive calls returned successful THEN
              put new in the arc-binding of dag2 in e;
       return (dag2);
  ELSE return ⊥ immediately
END;
```

This way, the structure-sharing scheme essentially avoids copying of nodes since original nodes are preserved using the skeleton and environment. However, the cost for these characteristics is rather steep. That is so because there is an inevitable drawback of this scheme

[2] In Pereira's data structure, forwarding is done to add the rerouting information to the environment *e*.

due to the *environment* storage scheme. The drawback is the fixed cost log(d) overhead that is required for *all* node accesses. The reason for the cost is that the information concerning each node is represented distributively in the *environment*; every time a node is accessed, the *environment* has to be looked up in order to update the *skeleton* associated with the node so that the required feature structure is dynamically created. In other words, for any operation accessing or manipulating a graph, there will be a fixed log(d) overhead (where d is the number of nodes in the graph) associated with each node in the entire graph in order to assemble the whole graph from the *skeleton* and the *environment*. Thus, although Pereira's scheme effectively avoids excessive copying, the trade-off required for his scheme is expensive.

## 3.2 Karttunen's method

Karttunen introduced an algorithm based upon the notion of *reversible unification* ([Karttunen, 1986b]). He reports in [Karttunen, 1986a] that this simple *reversible* method is more effective in reducing parsing time than the previous methods [Karttunen and Kay, 1985] and [Pereira, 1985]. The basic idea behind reversible unification is that when a destructive change is about to be made, the contents of the original graph are saved, then so that after a destructive unification, copies can be created from the result of the destructive unification and all destructive changes can be undone by restoring the graphs, using the information saved prior to the destructive operations. In his D-PATR implementation of reversible unification[3], Karttunen uses two arrays to save the original information. In one array, (pointers to) node structures are stored prior to a destructive operation. In another array, the actual content of the node structures, i.e., the attribute value pairs, are stored for each of the node structures (since the contents of the

[3] The discussion of Karttunen's method is based on the D-PATR implementation on Xerox 1100 machines ([Karttunen, 1986a]).

structures are going to be changed). After the top-level unification operation is done, the nodes are restored by setting the values saved in the array. Copies are made after a successful unification and only the necessary nodes are copied to create a new dag. Since Karttunen actually creates a copy after a successful unification (whereas in Pereira's scheme no copies are created and a dag is assembled every time it is needed), once the copy is created there will be no log(d) overhead for node accesses associated with Pereira's algorithm. On the other hand, there will be a cost of saving the dag structures and their values prior to destructive operations which is proportionate to the size of the input graph. There is also a cost of *reversing* the unification operation every time unification is performed which is also proportionate to the size of the input graph. Thus, if the size of the input graph grows then the cost of saving and reversing changes can be high.

## 3.3 Wroblewski's method

Wroblewski[1987] introduced a different scheme based upon the notion of *incremental copying.* His algorithm is known as "Wroblewski's nondestructive unification scheme" and has been considered as the fastest graph-unification algorithm. The basic idea behind his scheme is to create copies incrementally during unification only when such a need arises. It is a combination of a destructive unification algorithm unify1 (similar in its control structure to Pereira's algorithm) and a nondestructive algorithm unify2 in which copies are created incrementally. Unify1 is called only when either (or both) of the highest nodes of the input graphs are current copies of other nodes (so that they can be modified destructively without losing the original grammar and constituent graphs).

As a data structure, a node is represented with four fields: 'forward', 'arc-list', 'copy', and 'status'. 'The forward' field contains (a pointer to) another node which the node is being forwarded to. Arc-list contains the list of arc structures (i.e., mappings to other nodes). In addition to these two fields common in graph-based algorithms, Wroblewski's scheme has the added fields called 'copy' and 'status'. In 'copy' fields, the pointers to copy nodes that are created incrementally during unification are stored. 'Status' field contains a flag that indicates whether or not a particular node is part of an original graph (i.e., whether it is an original node or a copy of some other node). Later implementations of his algorithm (such as the one implemented in [Morimoto, *et al*, 1990]) use mark (or generation) field in place of status field. The mark field contains an integer which determines the currency of the copy node by comparing it to a global counter which is incremented every time the top level unification is called. The representation for an arc is a standard one. It is a pair of 'label' and 'value'. 'Label' contains an atomic symbol (i,e, a feature) which labels the arc, and 'value' contains another node structure.

```
        NODE
+---------------+
|   forward     |
+---------------+
|   arc-list    |          ARC
+---------------+    +---------------+
|     copy      |    |     label     |
+---------------+    +---------------+
|    status     |    |     value     |
+---------------+    +---------------+
```

Figure 3-3: **Wroblewski's Data Structure**

In Wroblewski's algorithm, unify2 first receives the input dags. If both of the highest nodes do not have copies then one copy node is created. This copy node is stored in the 'copy' field of the two input dag nodes. Also the 'status' of the copy node is set to be "copy". Then two set-complement operations are performed to produce two sets, i.e., *newd1* and *newd2*. *Newd1* contains the set-difference arcs of dag1 and dag2 (i.e., arcs with labels that exist in dag1 but not in dag2). *Newd2* contains the set-difference of dag2 and dag1. Also another set *shared* is created which contains a set-intersection of dag1 and dag2 arcs. Then for all arcs in the set *shared*, the destinations of the shared arcs from the dag1 and dag2 are recursively unified. Every time one recursion to a shared arc succeeds, the shared arc with the new value (result of recursion) is added to the copy node. If all recursion succeeds copies of arcs in both *newd1* and *newd2* are made while the 'copies' of nodes contained within the arcs are respected. The copies of the union of *newd1* and *newd2* are placed in arc-list of the copy node. This is the nondestructive incremental copying scheme in Wroblewski's unify2 algorithm. Also, if either (or both) highest input graph nodes is a copy, then all of this is bypassed and the destructive unify1 is called, adding the changes directly on the copy nodes.

Below is Wroblewski's Unify2. Unify1 is the same as Pereira's algorithm. The only difference from Pereira in Unify1 is that in Wroblewski's algorithm, forwarding is done by directly putting the forwarded node in the 'forward' field of a node instead of storing it in a *environment*. Also when complementarcs (dag1,dag2) are stored into dag2 after successful recursive calls, 'new' is stored directly into the 'arc-list' of dag2. Thus, Wroblewski's Unify1 is a destructive version of Pereira's algorithm. In the Wroblewski method, Unify1 is called only when either of the input graphs is a copy. Therefore, there will be no modification made to the original graph. Below is Wroblewski's Unify2, which copies incrementally while unification progresses:

WROBLEWSKI'S UNIFY2

```
FUNCTION unify2 (dag1-underef,dag2-underef);
  dag1 ← dereference(dag1-underef);
  dag2 ← dereference(dag2-underef);
  IF (dag1 $\equiv_r$ dag2) THEN
     return(dag2);
  ELSE IF (dag1 $\equiv_r$ Top) THEN
     forward(dag1,dag2);
     return(dag2);
  ELSE IF (dag2 $\equiv_r$ Top) THEN
     forward(dag1,dag2);
     return(dag1);
  ELSE IF (dag1 and dag2 are atomic and the values are equal) THEN
```

```
    forward(dag1,dag2);
    return(dag1);
ELSE IF (dag1.copy and dag2.copy are empty) THEN
    copy ← (create-node);
    copy.status ← "copy";
    dag1.copy ← copy;
    dag2.copy ← copy;
    newdag1 ← complementarcs(dag1,dag2);
    newdag2 ← complementarcs(dag2,dag1);
    shared ← intersectarcs(dag2,dag1);
    FOR EACH arc IN shared DO
        result ← unify2(destination of
                              the shared arc for dag1,
                        destination of
                              the shared arc for dag2);

        copy.arc-list ← result;
    FOR EACH arc IN (union newdag1,newdag2) DO
        recursively copy the value of each arc
        honoring existing copies within, and place
        this value in copy
    return(copy)
ELSE IF (dag1.copy xor dag2.copy is non-empty) THEN
    unify1(dag1.copy,dag2)
    return(dag1.copy);
ELSE IF (dag1.copy and dag2.copy are non-empty) THEN
    unify1(dag1.copy,dag2.copy)
END;
```

Below is an example of his nondestructive unification taken from [Wroblewski, 1987]. In the following series of figures, dashed lines indicate the contents of *copy* field. Darkened circles represent original input nodes and hollow circles represent nodes which are current copies. We quote (with slight modification in terminology) his explanations in order to walk through his unification algorithm.

"First figure shows (Figure 3-4) the state of unification after the path $< a,b >$ has bee followed during unification. Unify2 has recursed twice and returned to the top node; three new nodes have been created, one a copy of the root, one a copy of the node on the path $< a >$ and the last a copy of the node on the path $< a,b >$. The copy field of the appropriate nodes in dag1 and dag2 have been filled with the copy nodes, as indicated by the dashed lines."

"In Figure 3-5, Unify2 has follwed the path $< d >$ on the argument dags. But notice that the nodes at the end of path $< a >$ and at the end of path $< d >$ in dag2 are the same; a copy

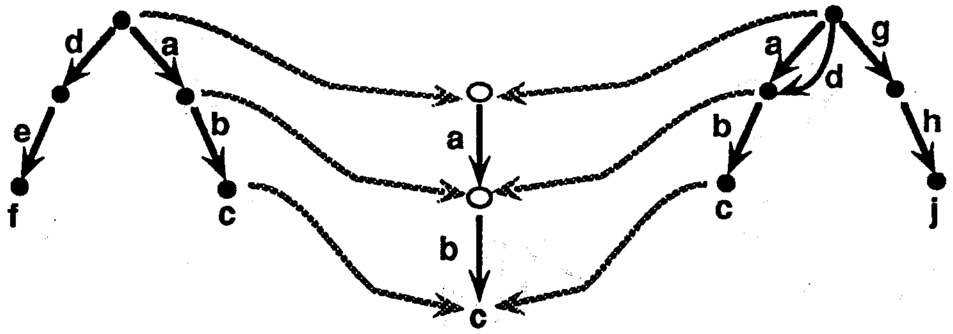


Figure 3-4: **Wroblewski's method: Snapshot 1**

of this node was previously made when traversing the path $< a, b >$, and so this copy is reused rather than allocating a new node. Subsequently, an arc labeled 'e' is added to this reused copy. Finally, Unify2 recursion unwinds back to the root node of both dags."

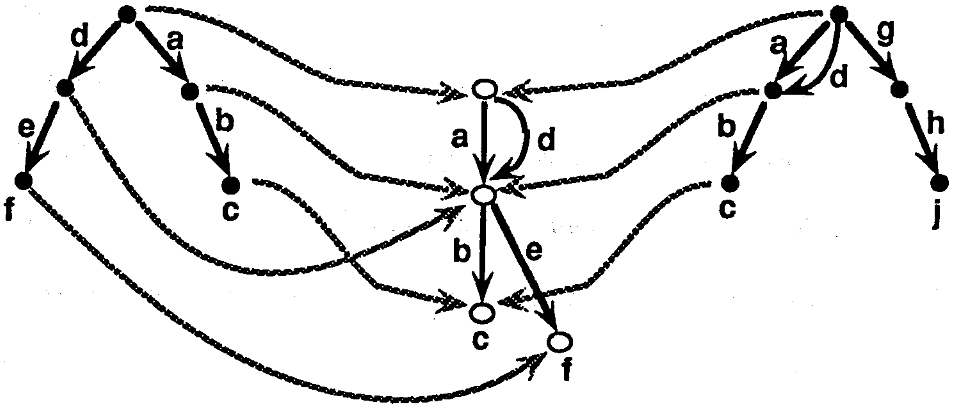


Figure 3-5: **Wroblewski's method: Snapshot 2**

"In Figure 3-6, Unify2 has added the arc labeled 'g' in dag2 to the result graph, making a copy of the subgraph at the end of that arc and placing it in the result graph. Notice that the subgraph dag2/$< g, h >$ was copied even though there existed no corresponding subgraph in dag1."

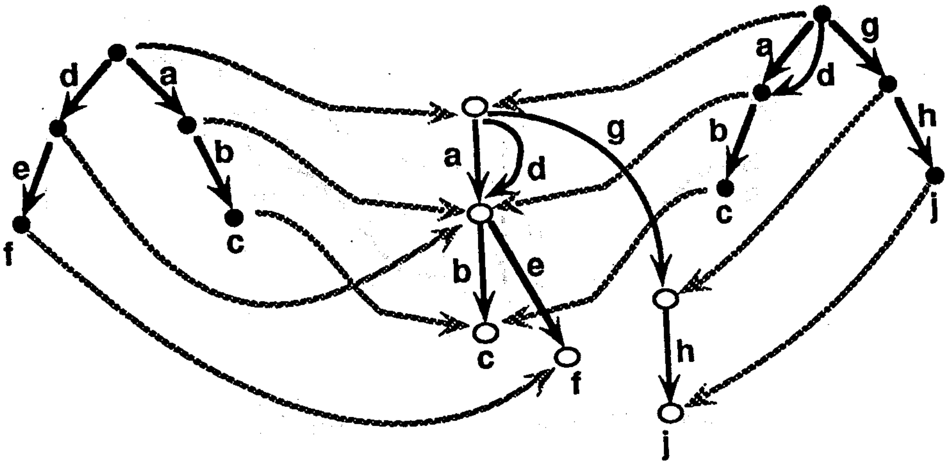


Figure 3-6: **Wroblewski's method: Snapshot 3**

As discussed by Wroblewski [1987] only 6 new nodes and 6 new arcs are created in the above unification. In a naiive destructive unification which uses Unify1, 10 nodes and 9 arcs would be created as copies of dag1 and dag2 prior to calling unify1. This way Wroblewski successfully reduced the number of wasteful operation (unnecessary copying) by introducing the incremental copying scheme. Since Wroblewski directly places nodes in the copy and arc-list of the nodes, no structure-sharing based on *environment* is performed. Therefore, there is no fixed cost log(d) overhead as associated with Pereira's algorithm for accessing the nodes. Also, changes in the copy field can be cancelled constant time by invalidating the copy field (for example, by using generation counters), therefore, there will be no cost for reversing the destructive changes which were associated with Karttunen's reversible unification.

Thus, the nondestructive unification algorithm was an immediate success and was immediately adopted by natural language research laboratories around the world. Of course there was some disadvantage to Wroblewski's scheme. Over-copying does occur in some cases, for example, in some configurations where dag1 contains a variable and dag2 contains a convergence on a variable ([Wroblewski, 1987]). Also, we will be discussing the inherent 'early copying' problem of the incremental copying scheme. However, despite the shortcomings of his methodology, until recently his method has been accepted as the most efficient method for graph unification. Later, Kogure and Kato ([Kogure, 1989]) developed a version of Wroblewksi's algorithm which extended it to handle cyclic feature structures. Their method was to check whether an arc with the same label already exists (i.e., so called "occur check") when an arc is added to a node. If such an arc already exists, destructive unification (unify1) is called for the destination of the existing arc unified against the destination of the node beeing added. If such an arc does not exist, the arc is simply added. Fortunately, their scheme for handling cycles in Wroblewski's framework is not costly (since it does not need to scan through the entire graph for occur check). Thus, Wroblewkski's method is also used by projects (such as ATR) that require cyclic feature structures.

## 3.4 Kogure's Method

The success of the incremental copying scheme proposed by Wroblewski led to a few research efforts based upon the incremental scheme. Among them, Godden ([Godden, 1990]) introduced a version of the incremental copying scheme in which he used a lazy evaluation technique for programming languages (Such as in *delay*, *force* in Scheme) and treated dags as active data structures. In Godden's method, evaluation for copying was delayed, using delayed streames, until a destructive change to the node is to be performed. At that point, copying is forced to perform the necessay copy operation of the original node. This way, copies are created incrementally during unification using lazy evaluation. Although using lazy evaluation to delay copying seemed a straightforward answer to some of the problems of Wroblewski's method, Godden's method was never favorable to efficient implementations of Wroblewski's original method. First of all, delayed evaluation is not a cheap operation. The cost of creating delayed closures is a potentially costly operation. Secondly, some of the burdens of excessive copying were simply replaced by creating closures using lazy evaluation. Since delayed closures may not be needed at the end of a unification to create a result graph, some closures delayed on a lazy stream simply get wasted. Thus, although, some copies that were originally wasted by creating structures for nodes were avoided, other wastes were produced by closures that were not forced.

Kogure [1990] introduced a different lazy incremental unification scheme by using dependency pointers instead of delayed closures. In his scheme, nodes to be copied contain a backward pointer (called *copy dependency* link) to the mother node of the graph, so that copies are not created from higher regions in the graph to the lower regions in the graph; instead, copies are created from the lower region in the graph. This scheme avoids copies of nodes whose subgraphs were never modified. As a result, an unmodified subgraph of the input graph is shared with the original input graph. Thus, Kogure introduced structure-sharing to the incremental copying scheme. He found that some subgraphs copied in Wroblewski's scheme did not actually need to be copied. Figure 3-7 is the example taken from [Kogure, 1990]:

Since the subgraphs E, F, D, in the figure were never modified, Wroblewski's scheme clearly overcopies them. Kogure introduced the *copy dependency* pointers stored in the nodes to ensure that copying of the nodes was delayed only until the children nodes were modified and virtually eliminated the redundant copying in Wroblewski's algorithm. Unfortunately, Kogure's method also has its trade-offs. They are due to the need to maintain the *copy dependency* pointers in each of the nodes in the entire graph. In Kogure's method, each node has an added field called the *copy-dependency* slot, in which the list of pairs of mothers and the arc structures connecting mothers to nodes are stored. Thus, in addition to the need for an added field, this list of the

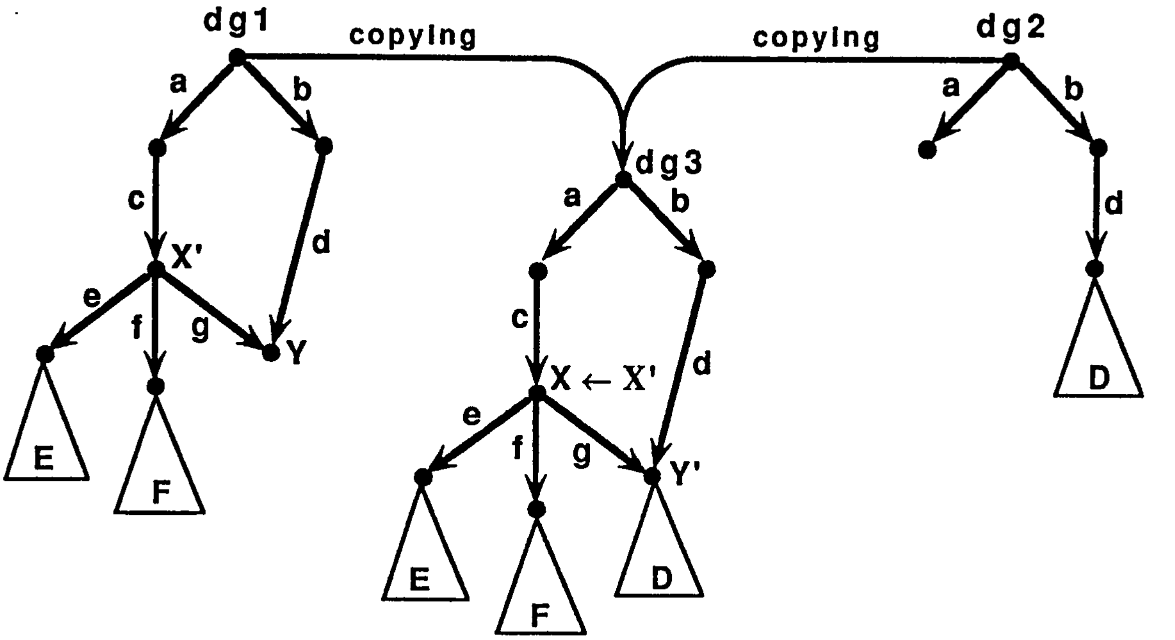


Figure 3-7: **Kogure's method avoids copying E,F,D.**

garbage-collectable additional 'conses' (node arc pairs) are stored in *copy-dependency.* Thus, by making incremental copying, in order to avoid redundant copying of unmodified subgraphs, Kogure had to introduce a bidirectionality in the entire directed graphs of feature structures; this could be steep if graphs are very large. Furthermore, there will be a need to traverse the graphs upward the dependency pointers to make the copies which in return may result in another traversal to make further copies. In other words, his algorithm can avoid redundant copying of unmodified subgraphs, but it will need to traverse twice on a modified subgraph — once to unify and once to copy, traversing the dependency pointer backwards. Therefore, in the worst case, his algorithm becomes a two-pass operation, when the advantage of incremental copying was that it was a one-pass operation (once to traverse down to unify and copy incrementally). Thus, Kogure's algorithm should be favorable to Wroblewski's when the input grammar contains large subgraphs which are rarely modified, then gains by structure-sharing of unmodified subgraphs more than offset the need for added data-structure and added garbage collections. However, if the grammar's behaviour was designed to modify graphs frequently, then the need for extra traversal can be considered very costly, making it disadvantageous compared to Wroblewski's original algorithm.

## 3.5 Emele's Method

Emele1991 also introduced a method based upon incremental copying and lazy operation of copying. He called his scheme Lazy Incremental Copying. His method is a combination of Wroblewski's incremental scheme and the structure-sharing idea of Pereira. The incremental copying algorithm itself is similar to Wroblewski's; however, lazy copying is introduced to delay the copying of nodes so that copying is done only when the destructive change is about to happen. Godden used delayed evaluation for delaying copying, and Kogure used copy-dependency pointers to delay copying. Emele uses a series of what he calls chronological dereference chains in order to make delayed copying possible. In order to evaluate Emele's scheme, we need to separate the incremental copying part of his algorithm and the data structure based upon chronological derefence chains which resembles the last-call optimization technique of Prolog (as in [Warren, 1983]). As is the case with Pereira's environment and skeleton method, Emele's scheme for chronological dereference is a data-structure technique and is independent of the actual unification algorithm itself. In other words, it is possible to combine Emele's chronological dereference scheme with any other unification algorithms including the one we are proposing in the next chapter in this thesis. Whether such a combination is a good idea or not is a different question. It would probably depend on the kind of grammar with which that unification is intended to be used.

What Emele does in his chronological dereference scheme is to adopt Pereira's structure-sharing idea, but instead of using a global branch environment, each node records its own environment. The chronological dereference is performed by following the chain of forwarding pointers based upon the *environment* list which decides whether a forwarding pointer should be followed or not. An *environment* is represented as an ordered sequence of valid generation counters (such as $< 1, 2, 3, 4, 5, 6... >$). The current generation is defined as the last element in this sequence.

Figure 3-8 taken from [Emele, 1991] is an example of a chronological dereference chain. It illustrates how dereference works with respect to the environment: "Node b is the class representative for environment $< 0 >$, node c is the result of dereferencing for environments $< 0, 1 >$ and $< 0, 1, 2 >$, and finally node f corresponds to the representative for the environment $< 0, 1, 2, 3 >$ and all further extentions that did not add a new forwarding pointer to newly created copy nodes".

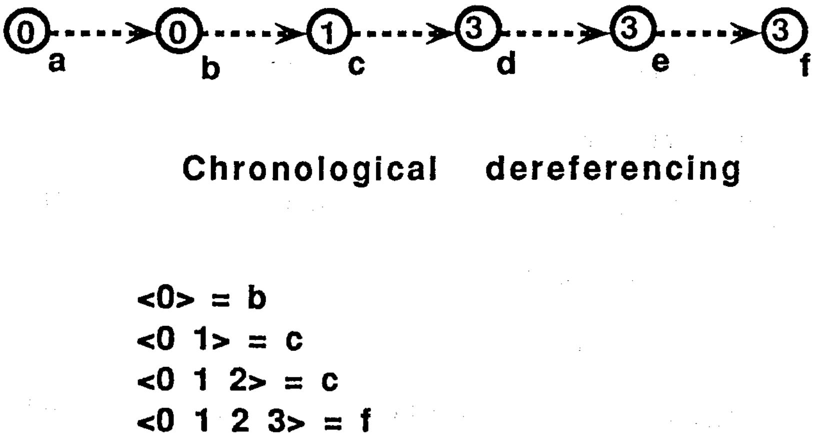


Figure 3-8: **Traversing forwarding links according to *environment***

With this method of chronological dereferencing and locally represented environment, Emele effectively attained a data structure that supports structure-sharing and avoids the potentially complex operation of merging environments in Pereira's structure-sharing scheme. He combined this data-structure with Wroblewski's incremental copying scheme and called it a "Lazy Incremental Copying" scheme.

What follows is a walk-through of how this happens in his Lazy Incremental Copying scheme using the example input graphs below (Figure 3-9).

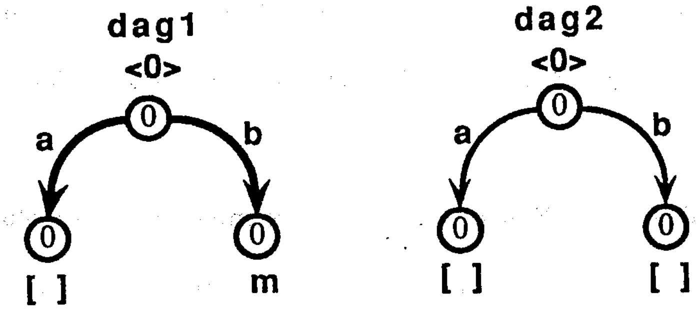


Figure 3-9: **Sample input graph**

Figure 3-10 shows that copies with generation 1 are created incrementally while unification progresses. Since everything needs to be copied there is no structure-sharing of nodes. Note that in Emele's scheme, as in Pereira's scheme, arcs are not copied. Instead the result graph dag3 comes with the *environment* $< 0, 1 >$ so that subgraphs of dag3 are created by looking at the environment and traversing down dagl.

Figure 3-11 shows the unification of the result graph dag3 with a new graph dag4.

Note that arcs (a 0) and (b 0) of dag3 are simply placed in dag5. Some copies are mad( (the ones numbered with generation 2) since destructive changes were to be made. After the unification dag5 can be constructed by following the original graphs stored in dag3 (which is actually dagl) and by performing the chronological dereferencing on the subgraphs.

This way, Emele successfully combined Wroblewski's incremental scheme with Pereira's structure sharing scheme to combine the advantages of the two. On the other hand, he also

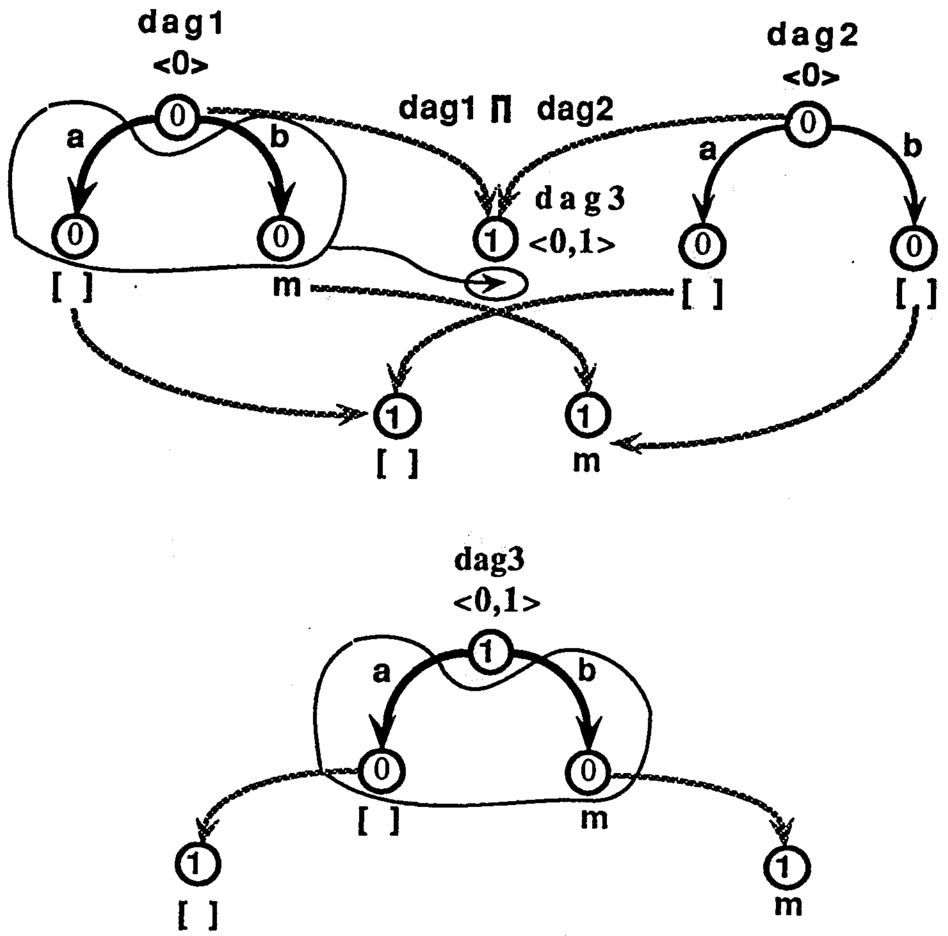


Figure 3-10: **dag3 = dag1 ⊓ dag2**

combined the disadvantages as well. The disadvantages of Wroblewski's incremental scheme, which is inherent in incremental method and is shared by Godden, Kogure, and Emele, will be discussed in the next chapter. The disadvantage of the structure-sharing scheme is that there will be a cost for sharing structures which could be expensive. In Emele's method this shows as the cost for traversing the chronological dereference chain. As we have seen in the above example, every time a destructive change is to be made to a node, a copy of the node is created and put at the end of the chronological dereference chain. Figure 3-12 is a picture of dag5 from the above example.

After only two successful unifications, with a graph containing only 3 non-root nodes and 3 arcs, we need to follow the derefence chain 4 times. Since every unification in a shared arc is a destructive operation, this dereference chain is extended every time a unification is performed

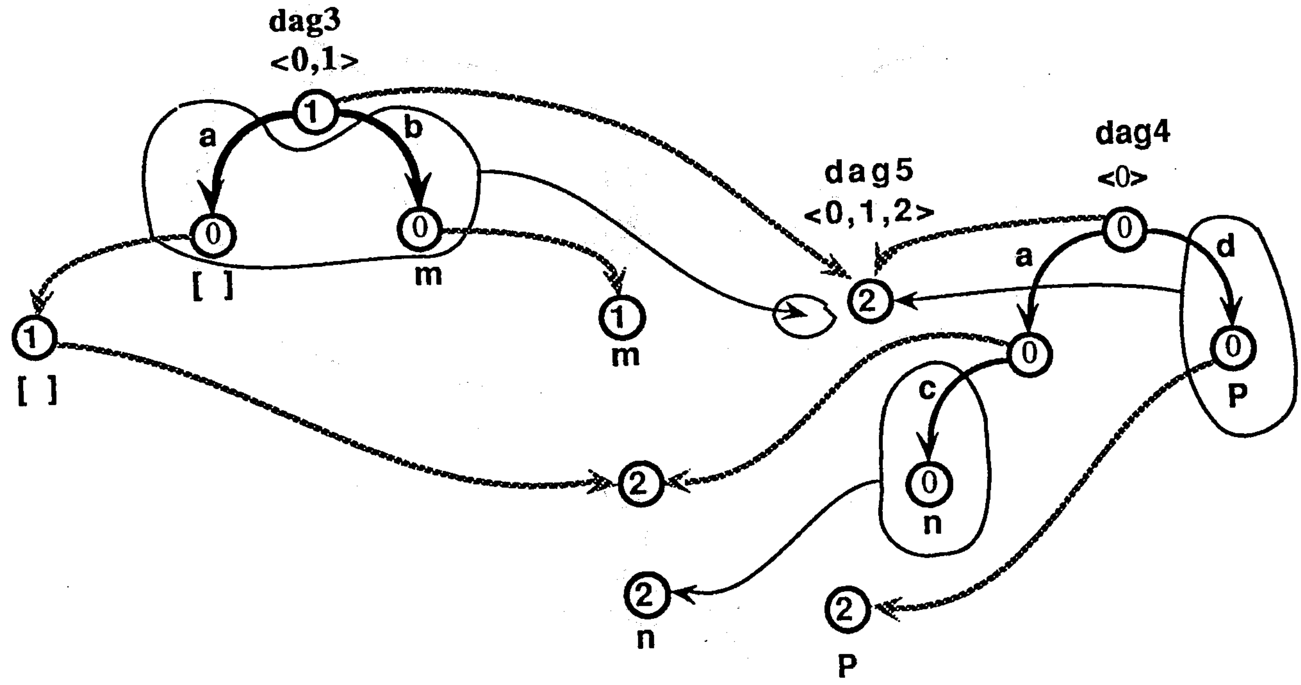


Figure 3-11: dag5 = dag3 ⊓ dag4

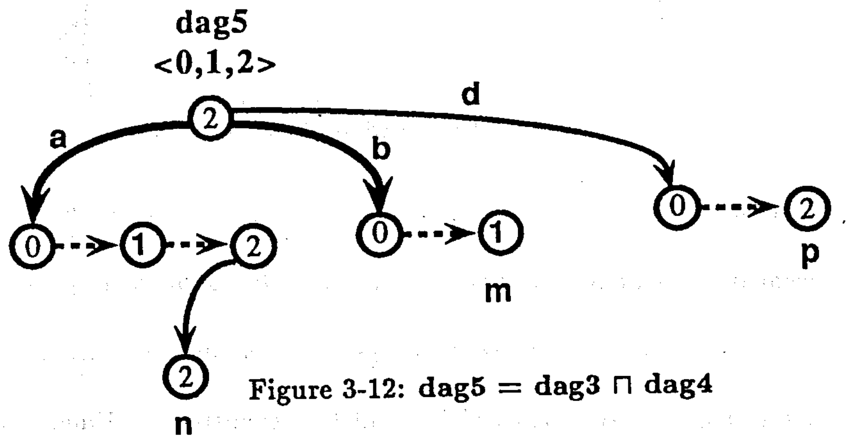


Figure 3-12: dag5 = dag3 ⊓ dag4

in a shared arc. Since some constituent graphs are unified a great number of times in typical large scale natural language systems, the cost of this can be very high. A long chronological dereference chain may be needed to be followed in order to get the node that is needed. Note that the chain must be followed every time the graph is needed for each and every node in the entire graph. Since each constituent graph built during unification can grow extremely large in large scale systems, and since unification between them escalates the complexity of traversing the chronological dererence chain, the question of whether the introduction of this data structure scheme is desirable is an open question dependent upon application areas. If we could find an application where graphs contain only few shared arcs and where graphs are rarely reused once unified, then Emele' scheme could be an ideal option. In such a case, as we have discussed above, his scheme can be adopted to any existing unification algorithms, including the one we are proposing in this thesis.

# Chapter 4

# Quasi-Destructive Graph Unification

## 4.1 Introduction

In designing an efficient graph unification algorithm, we have made the following observation which influenced the basic design of the new algorithm described in this thesis:

**Unification does not always succeed.**

In a typical natural language system with a relatively small grammar size, 60 to 80 percent of unifications attempted during a successful parse result in failure. As the grammar size increases, the number of unification failures for each successful parse increases. For example, in our large-scale speech-to-speech translation system jointly under development by CMU and ATR Interpreting Telephony Research Laboratories, we estimate more than 90% of unifications to be failures during a successful parse. If a unification fails, any computation performed and memory consumed during the unification is wasted.

Another observation about the behaviour of graph unification which seems well accepted in the existing literature is that:

**Copying is an expensive operation.**

Copying a node places a heavy burden on the parsing system. Wroblewski[1987] calls it a "computational sink". Copying is expensive in two ways: 1) it takes time; 2) it takes space. Copying takes time and space essentially because the area in the random access memory needs to be dynamically allocated, which is an expensive operation. We calculated the computation time cost of copying to be more than 90 percent of the total parsing time in our large-scale speech-to-speech translation system. This time/space copying burden presents problems in an environment where computational resources are limited due to the size of the grammar and other knowledge sources. (Also, the creation of unnecessary copies eventually triggers garbage collections more often in a Lisp environment, which also degrades the overall performance of the parsing system.) In general, parsing systems (such as large LR tables of Tomita-LR parsers, expanding tables and charts of Earley, and active chart parsers) are always short of memory space. Our own phoneme-based generalized LR parser for speech input is always running on a swapping space because the LR table is too big, and the marginal addition or subtraction of memory space consumed by other parts of the system often has critical effects on the performance of these systems. An experiment conducted at ATR showed that in order to attain a stable performance of a parser, a physical memory space required for the sentence that requires the most memory needs to be guaranteed to the system. We have seen that the amount of memory (conses) consumed by copying operations during a parse determines the necessary physical memory.[1] With the aforementioned observations, we propose the following

[1] For example, as we will see from the data in Chapter 6, the memory needed for Wroblewski's algorithm was 5 to 6 times greater than the proposed scheme. This means that not only were sentences faster with our scheme, but also that some sentences could not be parsed at all using Wroblewski's algorithm on our machine environment due to the physical limit of memory speed.

principles to be the desirable conditions for an efficient graph unification algorithm:

- **Copying should be performed only for successful unifications.**
- **Unification failures should be found as soon as possible.**

By way of definition, we would like to categorize excessive copying of graphs into Over Copying and Early Copying. Wroblewski[1987] also defines Over Copying and Early Copying. Our definition of over copying is the same as Wroblewski's; however, our definition of early copying is slightly different.

- **Over Copying:** Two graphs are created in order to create one new graph. This typically happens when copies of two input graphs are created prior to a destructive unification operation to build one new graph.
- **Early Copying:** Copies are created prior to the failure of unification so that copies created since the beginning of the unification up to the point of failure are wasted.

Wroblewski defines Early Copying as follows: "The argument dags are copied *before* unification started. If the unification fails then some of the copying is wasted effort." He restricts early copying to cases that only apply to copies that are created prior to a unification. Restricting early copying to represent copies that are created prior to a unification leaves a number of wasted copies that are created during the same unification up to the point of the detection of failure. Therefore, these wasted copies will not be covered by either of the above two definitions for excessive copying. We would like Early Copying to mean all copies that are wasted due to a unification failure, whether these copies are created before or during the unification.

## 4.2 The Quasi-Destructive Graph Unification Algorithm

We would like to introduce an algorithm which addresses the criteria for fast unification discussed in the previous sections ([Tomabechi, 1991a]). It also handles cycles without over copying (without any additional schemes such as those introduced by Kogure[1989]).

As a data structure, a node is represented with six fields: 'type', 'arc-list', 'comp-arc-list', 'forward', 'copy'[2], and generation.[3] The data-structure for an arc has two fields, 'label' and 'value'. 'Label' is an atomic symbol which labels the arc, and 'value' is a pointer to a node structure.

```
        NODE
+---------------+
|     type      |
+---------------+
|   arc-list    |
+---------------+
| comp-arc-list |
+---------------+
|   forward     |             ARC
+---------------+      +---------------+
|    copy       |      |     label     |
+---------------+      +---------------+
|  generation   |      |     value     |
+---------------+      +---------------+
```

Figure 4-1: **Node and Arc Structures**

The central notion of the Q-D algorithm is the dependency of the representational content on the global timing clock (or the global counter for the current generation of unifications). Any modification made to comp-arc-list, forward, or copy fields during one top-level unification can be invalidated by one increment operation on the global timing counter. Contents of the

[2] Martin Emele of University of Stuttgart suggested that a separate field for 'copy' may be saved by using a forward link only, since copy link is needed only when forward link is not used.

[3] Note that [Tomabechi, 1991a] used separate mark fields for the comp-arc-list, forward, and copy; currently however, only one generation mark is used for all three fields. Thanks are due to Hidehiko Matsuo of Toyo Information Systems for suggesting this.

comp-arc-list, forward and copy fields are respected only when the generation mark of the particular node matches the current global counter value. Q-D graph unification has two kinds of arc lists: 1) arc-list and 2) comp-arc-list. Arc-list contains the arcs that are permanent (i.e., ordinary graph arcs) and comp-arc-list contains arcs that are valid only during one top-level unification operation. The algorithm also uses two kinds of forwarding links, i.e., permanent and temporary. A permanent forwarding link is the usual forwarding link found in other algorithms ([Pereira, 1985], [Wroblewski, 1987], etc). Temporary forwarding links are links that are valid only during one top-level unification. The currency of the temporary links is determined by matching the content of the generation field for the links with the global counter; if they match, the content of this field is respected[4].

As in Pereira[1985], the Q-D algorithm has three types of nodes: 1) :atomic, 2) :Top[5], and 3) :complex. The :atomic type nodes represent atomic symbol values (such as 'Noun'), :Top type nodes are variables, and :complex type nodes are nodes that have arcs coming out of them. Arcs are stored in the arc-list field. The atomic value is also stored in the arc-list if the node type is :atomic. :Top nodes succeed in unifying with any nodes and the result of unification takes the type and the value of the node with which the :Top node was unified, :atomic nodes succeed in unifying with :Top nodes or with :atomic nodes with the same value (stored in the arc-list). Unification of an :atomic node with a :complex node immediately fails. :complex nodes succeed in unifying with :Top nodes or with :complex nodes whose subgraphs all unify.[6] What follows are the central quasi-destructive graph unification algorithm and the

[4] We do not have a separate field for temporary forwarding links; instead, we designate the integer value 9 to represent a permanent forwarding link. We start incrementing the global counter from 10 so that whenever the generation mark is not 9, the integer value must equal the global counter value in order to respect the forwarding link.

[5] We called this :bottom in [Tomabechi, 1991a] and [Tomabechi, 1992]. Also it is called leaf in Pereira's algorithm.

[6] Arc values are always nodes and never symbolic values because :atomic and :Top nodes may be (or may become) pointed to by multiple arcs depending on grammar constraints. We do not want arcs to contain

dereferencing[7] function. Following that is the algorithm description for copying nodes and arcs (called from unify0) while respecting the contents of comp-arc-lists.

terminal atomic values.

[7] Dereferencing is the operation of recursively traversing forwarding links to return the target node of the forwarding (as presented in discussions of Pereira's and Wroblewski's algorithms in Chapter 3).

## Quasi-Destructive Graph Unification

```
FUNCTION unify-dg(dg1,dg2);
  result ← catch with tag 'unify-fail
            calling unify0(dg1,dg2);
  increment *unify-global-counter*; ;; starts from 10 [8]
  return(result);
END;

FUNCTION unify0(dg1,dg2);
  if '*T* = unify1(dg1,dg2); THEN
    copy ← copy-dg-with-comp-arcs(dg1);
    return(copy);
END;

FUNCTION unify1 (dg1-underef,dg2-underef);
  dg1 ← dereference-dg(dg1-underef);
  dg2 ← dereference-dg(dg2-underef);
  IF (dg1.copy is non-empty) THEN
     dg1.copy ← nil; ;; cutoff uncurrent copy
  IF (dg2.copy is non-empty) THEN
     dg2.copy ← nil;
  IF (dg1 $\equiv_r$ dg2)[9]THEN
     return('*T*);
  ELSE IF (dg1.type = :Top) THEN
     forward-dg(dg1,dg2,:temporary);
     return('*T*);
  ELSE IF (dg2.type = :Top) THEN
     forward-dg(dg2,dg1,:temporary);
     return('*T*);
  ELSE IF (dg1.type = :atomic AND
              dg2.type = :atomic) THEN
              IF (dg1.arc-list = dg2.arc-list)[10]THEN
                 forward-dg(dg2,dg1,:temporary);
                 return('*T*);
              ELSE throw[11]with keyword 'unify-fail;
  ELSE IF (dg1.type = :atomic OR
              dg2.type = :atomic) THEN
              throw with keyword 'unify-fail;
  ELSE shared ← intersectarcs(dg1,dg2);
       FOR EACH arc IN shared DO
                unify1(destination of
                              the shared arc for dg1,
                        destination of
                              the shared arc for dg2);
       forward-dg(dg2,dg1,:temporary);[12]
       new ← complementarcs(dg2,dg1);[13]
       IF[14](dg1.comp-arc-list is non-empty) THEN
             IF (dg1.generation = *unify-global-counter*) THEN
                  FOR EACH arc IN new DO
                           push arc to dg1.comp-arc-list;
             ELSE dg1.comp-arc-list ← nil;
       ELSE dg1.generation ← *unify-global-counter*;
            dg1.comp-arc-list ← new;
       return ('*T*);
```

```
END;
```

### Graph Node Dereferencing

```
FUNCTION dereference-dg(dg);
  forward-dest ← dg.forward;
  IF (forward-dest is non-empty) THEN
     IF (dg.generation = *unify-global-counter* OR
          dg.generation = 9) THEN
          return(dereference-dg(forward-dest));
     ELSE dg.forward ← nil;
            return(dg);
  ELSE return(dg);
END;
```

The functions Complementarcs(dg1,dg2) and Intersectarcs(dg1,dg2) return the set-difference (the arcs with labels that exist in dg1 but not in dg2) and intersection (the arcs with labels that exist both in dg1 and dg2). During the set-difference and set-intersection operations, the content of comp-arc-lists are respected as parts of arc lists if the generation mark matches the current value of the global timing counter. Forward(dg1, dg2, :forward-type) puts (the pointer to) dg2 in the forward field of dg1. If the keyword in the function call is :temporary, the current value of the *unify-global-counter* is written in the generation field of dg1. If the keyword is :permanent, 9 is written in the generation field of dg1.[15] The temporary forwarding links are necessary to handle reentrancy and cycles. As soon as unification (at any level of recursion through shared arcs) succeeds, a temporary forwarding link is made from dg2 to dg1 (dg1 to

[8] 9 indicates a permanent forwarding link.

[9] As discussed previously, this represents 'equal' in the 'eq' sense. Because of forwarding and cycles, it is possible that dg1 and dg2 are 'eq'.

[10] Arc-list contains atomic value if the node is of type :atomic.

[11] Catch/throw construct; i.e., immediately return to *unify-dg*.

[12] This will be executed only when all recursive calls into unify1 have succeeded. Otherwise, a failure would have caused an immediate return to *unify-dg*.

[13] Complementarcs(dg2,dg1) was called before unify1 recursions in [Tomabechi, 1991a], Currently it is relocated to after all unify1 recursions successfully return. Thanks are due to Marie Boyle of the University of Tuebingen for suggesting this.

[14] This check was added after [Tomabechi, 1991a] to avoid over-writing the comp-arc-list when it is written more than once within one unify0 call. Thanks are due to Peter Neuhaus of Universität Karlsruhe for reporting this problem.

[15] The Q-D algorithm itself does not require any permanent forwarding; however, the functionality is added because some grammar reader modules that read the path equation specifications into directed graph feature-structures use permanent forwarding to merge the additional grammatical specifications into a graph structure.

dg2 if dg1 is of type :Top). Thus, during unification, a node already unified by other recursive calls to unify1 within the same unify0 call has a temporary forwarding link from dg2 to dg1 (or dg1 to dg2). As a result, if this node becomes an input argument node, dereferencing the node causes dg1 and dg2 to become the same node and unification immediately succeeds. Thus, a subgraph below an already unified node will not be checked more than once even if an argument graph has a cycle.[16]

### QUASI-DESTRUCTIVE COPYING

```
FUNCTION copy-dg-with-comp-arcs(dg-underef);
  dg ← dereference-dg(dg-underef);
  IF (dg.copy is non-empty AND
      dg.copy.generation[17]= *unify-global-counter*) THEN
      return(dg.copy);[18]
  ELSE IF (dg.type = :atomic) THEN
      newcopy ← create-node();[19]
      newcopy.type ← :atomic;
      newcopy.arc-list ← dg.arc-list;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;
      return(newcopy);
  ELSE IF (dg.type = :Top) THEN
      newcopy ← create-node();
      newcopy.type ← :Top;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;
      return(newcopy);
  ELSE
      newcopy ← create-node();
      newcopy.type ← :complex;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;[20]
      FOR ALL arc IN dg.arc-list DO
          newarc ← copy-arc-and-comp-arc(arc);
          push newarc into newcopy.arc-list;
      IF (dg.comp-arc-list is non-empty AND
          dg.generation = *unify-global-counter*) THEN
              FOR ALL comp-arc IN dg.comp-arc-list DO
                  newarc ← copy-arc-and-comp-arc(comp-arc);
                  push newarc into newcopy.arc-list;
      dg.comp-arc-list ← nil;
      return (newcopy);
END;
```

[16] Also, during copying subsequent to a successful unification, two arcs converging into the same node will not cause overcopying simply because if a node already has a copy then the copy is returned.

```
FUNCTION copy-arc-and-comp-arc(input-arc);
  label ← input-arc.label;
  value ← copy-dg-with-comp-arcs(input-arc.value);
  return a new arc with label and value;
END;
```

Let us walk through a simple unification example first. What follows in the following two pages is simple unification of two graphs dg1 and dg2 which represent feature structures:

```
dg1
[[a S]
 [b []]]

dg2
[[a X01 []]
 [b X01]
 [c t]]
```

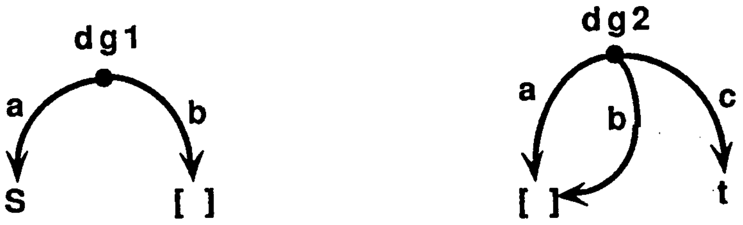
First, top-level unify-dg calls unify0 which in turn calls unify1. Unify0 will perform quasi-destructive copying operation after the top level call to unify1 successfully returns. Now top-level unify1 finds that each of the input graphs has arcs with labels a and b (*shared*). For now we represent arcs with label a as arc-a. Then unify1 is recursively called (unify1(2,5)). At step two, the recursion into arc-a locally succeeds, and a temporary forwarding link with time-stamp(n) is made from node 5 to node 2. At the third step (recursion into arc-b), by the previous forwarding to node 2, node 5 already has the value $S$ (by dereferencing). Then this unification returns a success and a temporary forwarding link with time-stamp(n) is created from node 3 to node 2. At the fourth step, since all recursive unifications (unify1s) into shared

[20] I.e., the 'generation' field of the node stored in the 'copy' field of the 'dg' node. The algorithm described in [Tomabechi, 1991a] used 'copy-mark' field of 'dg'. Currently 'generation' field replaces the three mark field described in the article.

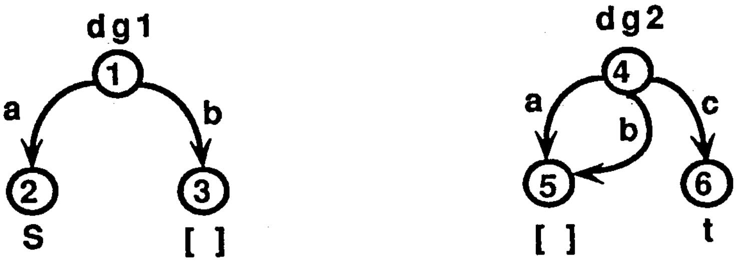
[21] I.e., the existing copy of the node.

[22] Creates an empty node structure.

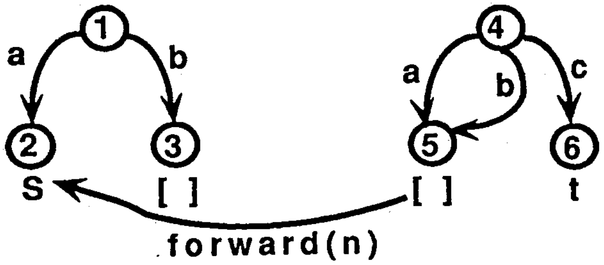
[23] This operation to set a newly created copy node into the 'copy' field of 'dg' was done after recursion into subgraphs in the algorithm description in [Tomabechi, 1991a] which was a cause of infinite recursion with a particular type of cycles in the graph. By moving up to this position from after the recursion, such a problem can be effectively avoided. Thanks are due to Peter Neuhaus for reporting the problem.



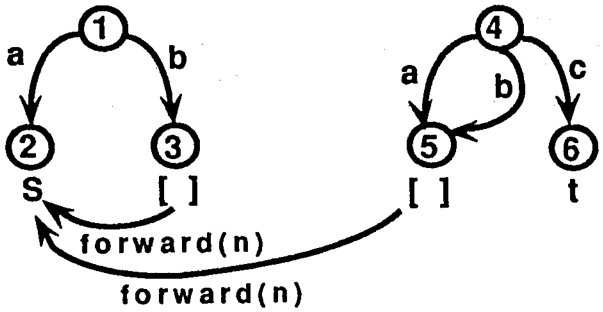
Trace of unify-dg(dg1,dg2) below:



```
          dg1  dg2
--- unify-dg( ①, ④ )                    generation n
    --- unify0( ①, ④ )
        --- unify1( ①, ④ )
            SHARED={( a , ② ),( b ,③ )}
             for each arc in shared
            --- unify1( ②, ⑤ )
                --- forward( ⑤ to② ) with generation n
```

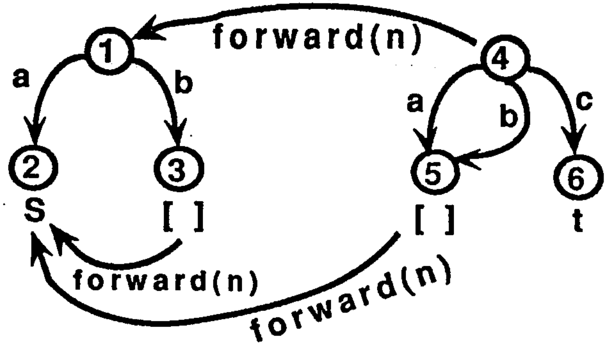


```
            --- EXIT unify1  *T*
            --- unify1( ③, ⑤ )
                --- By dereferencing, ⑤ is actually② since it was
                    forwarded above, since ③ is [ ], unification
                    immediately succeeds and forward ③ to② .
```



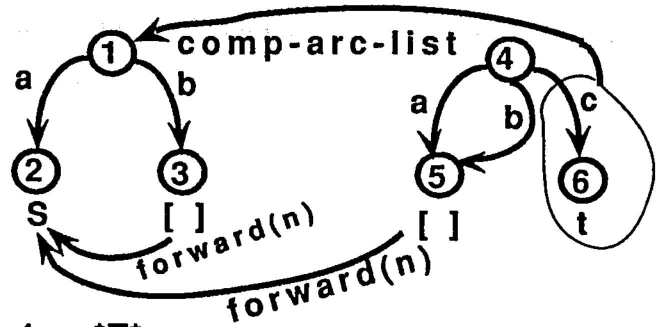
---- EXIT unify1 *T*

forward ④ to ① with generation $n$



NEW={( c , ⑥ )}

new is put in comp-arc-list of ① .

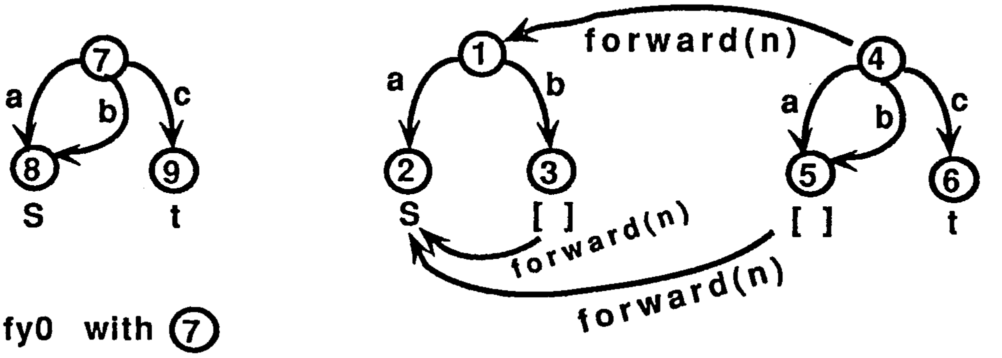


---- EXIT unify1 *T*

---- copy-dg-with-comp-arcs( ① )

---- make a copy of dg1 recursively respecting current forwarding and respecting valid comp-arc-list.

copy of ① at generation $n$

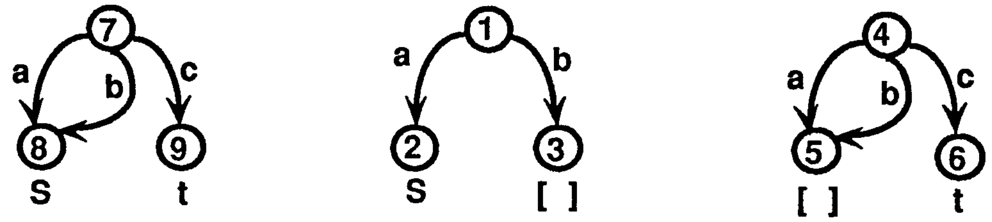


--- return unify0 with ⑦

--- increment generation

--- return ⑦

generation $n+1$



arcs succeeded, top-level unify1 creates a temporary forwarding link with time-stamp(n) from dg2's root node 4 to dg1's root node 1, and sets arc-c (*new*) into comp-arc-list of dg1 and returns success ('*T*). At the fifth step, a copy of dg1 is created respecting the content of comp-arc-list and dereferencing the valid forward links. This copy is returned as a result of unification. At the last step (step six), the global timing counter is incremented (n ⇒ n+1). After this operation, temporary forwarding links and comp-arc-lists with uncurrent time-stamp ($\neq$ n+1) will be ignored. Therefore, the original dg1 and dg2 are recovered in constant time without a costly reversing operation. (Also, note that recursions into shared-arcs can be done in any order, producing the same result).

As we just saw, the algorithm itself is simple. The essential difference between our unify1 and the previous ones such as Pereira's is that our unify1 is non-destructive. That is so because the complementarcs(dg2,dg1) are set to the comp-arc-list of dg1 and not into the arc-list of dg1. Thus, as soon as we increment the global counter, the changes made to dg1 (i.e., addition of complement arcs into comp-arc-list) vanish. As long as the generation value matches that of the global counter, the content of the comp-arc-list can be considered a part of arc-list and therefore, dg1 is the result of unification. Hence the name quasi-destructive graph unification. In order to create a copy for subsequent use, we only need to make a copy of dg1 before we increment the global counter, while respecting the content of the comp-arc-list of dg1.

This way, instead of calling other unification functions (such as unify2 of Wroblewski) for incrementally creating a copy node during a unification, we need only to create a copy after unification. Thus, if unification fails, no copies are made at all (as in Karttunen's scheme). Because unification that recurses into shared arcs carries no burden of incremental copying (i.e., it simply checks whether nodes are compatible), as the depth of unification increases (i.e., as the graph gets larger) the speed-up of our method should become conspicuous if a unification eventually fails. Since a parse that does not fail on a single unification is unrealistic, the gain from our scheme should depend on the number of unification failures that occur during a unification. As the grammar size increases, the number of failures per parse tend to increase and the graphs that failed get larger, and the speed-up from our algorithm should become more apparent. Therefore, the characteristics of our algorithm seem desirable.

What follows is a sequence of examples showing the way that temporary forwarding and comp-arc-list work to perform efficient unification. The quasi-destructive copying after unification copies the dg1s by simply following temporary forwarding pointers. Unlike Emele's method, the temporary forwarding does not glow since there is no chronological derefencing.

After a successful unification, one increment in the global counter invalidates all changes made to the graph.

First we will start with another simple example as shown in Figure 4-2 and Figure 4-3. Note that unify-dg(dg1,dg2) and unify-dg(dg2,dg1) get the same results. The result should be as in Figure 4-4. We can see that only a minimum number of copies are created.

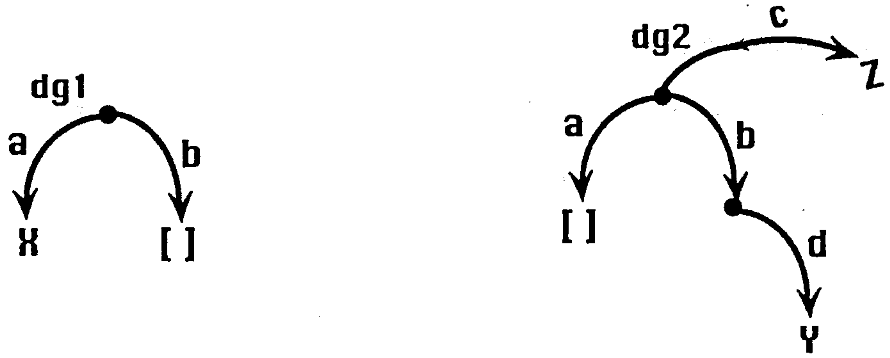


Figure 4-2: Another simple example

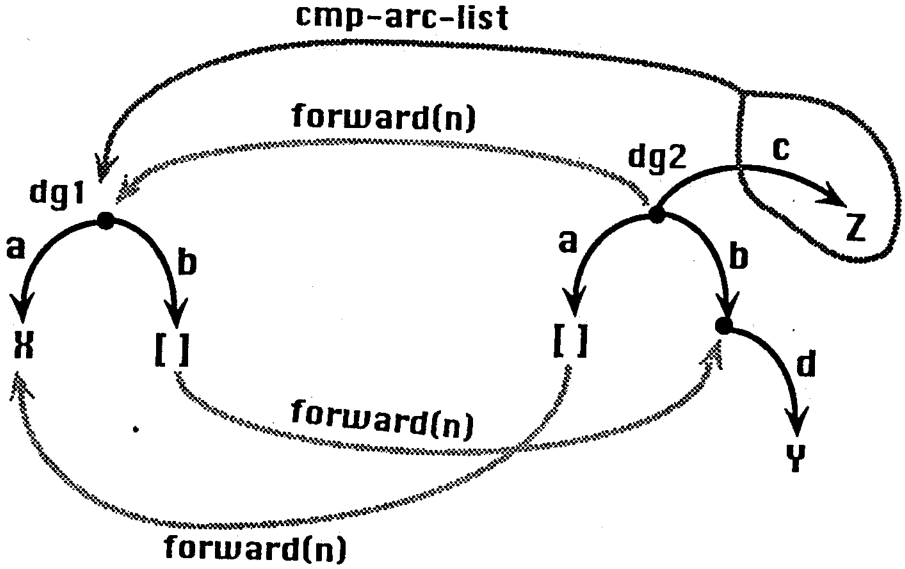


Figure 4-3: At the end of time n

Now, for a bit more complicated example (Figure 4-5, Figure 4-6), but one that works in a similar manner.

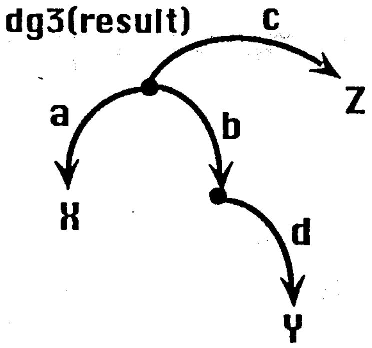


Figure 4-4: and the result

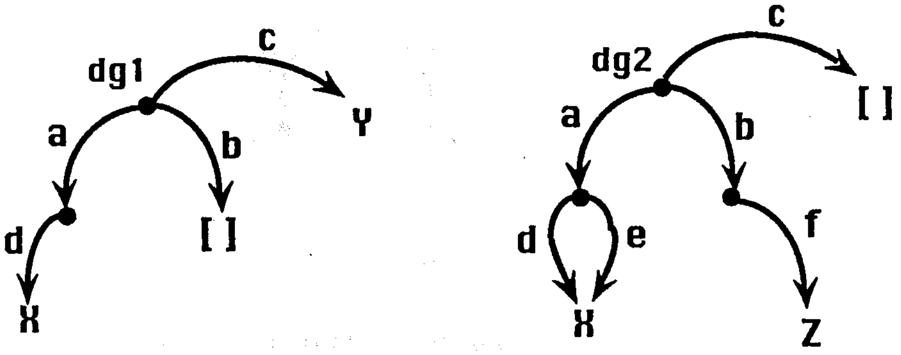


Figure 4-5: A little more difficult example

And the result in Figure 4-7

Now comes a difficult example. Not only might its workings be difficult to follow, it was impossible for most past unification algorithms. But if you follow the simple rule of traversing the arcs, and if you find [], just forward it to the counterpart. Add the complement arcs into comp-arc-list, and it turns out that the unification is rather straightforward in the QD framework. We will use four figures (Figures 4-8, 4-9, 4-10, and 4-11) to depict this one. The first step is to forward the dg2/$< a >$ which is [] to dg1/$< a >$ (Fig 4-9).

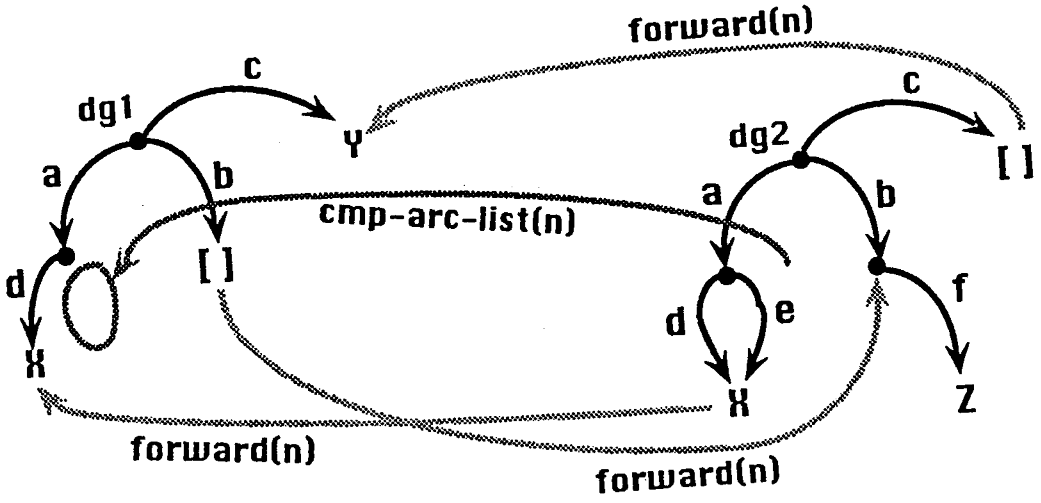


Figure 4-6: **At the end of time n**

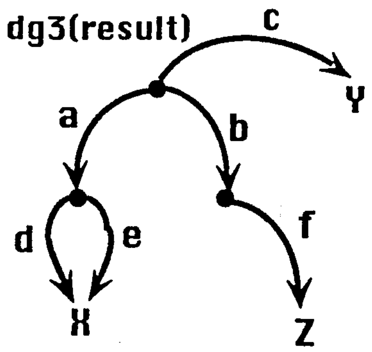


Figure 4-7: **and the result**

Then we unify dg1/$< b, d >$ and dg2/$< b, d >$. Since both are [], dg1/$< b, d >$ gets forwarded to dg2/$< b, d >$ (see the algorithm). Now we traverse into arc-c and unify dg1/$< c >$ and dg2/$< c >$. We find that dg1/$< c >$ is already forwarded to dg2/$< b, d >$ so actually we are unifying the dg2/$< b, d >$ with dg2/$< c >$. Since dg2/$< b, d >$ then is [], it succeeds and dg2/$< b, d >$ is forwarded to dg2/$< c >$. This is the end of the recursions into the shared arcs. Now, arc-f is the complementarc(dg2,dg1) therefore, it is put into the comp-arc-list of dg1. This is the end of the recursive calls to unify1 (Figure 4-10).

Now unify1 returns and unify0 makes a copy of dg1 respecting the current forwarding and

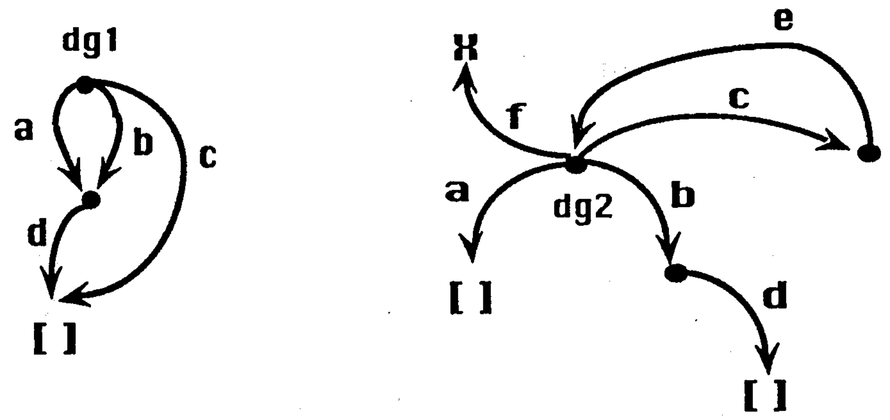


Figure 4-8: **A difficult example containing a cycle**

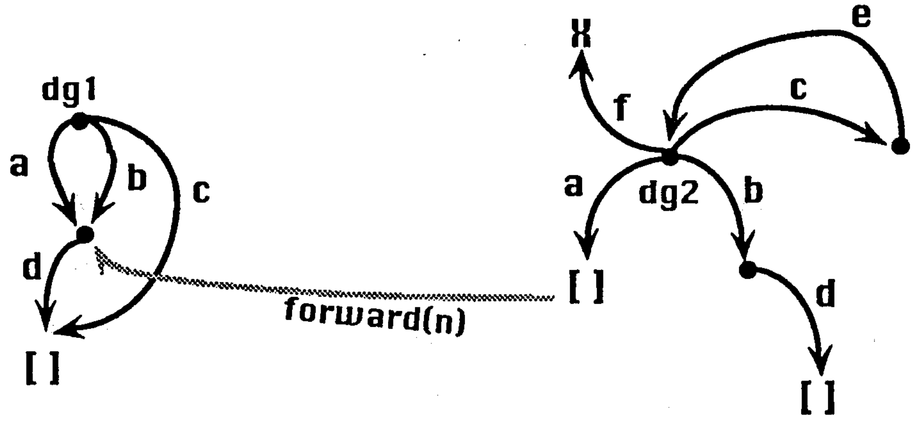


Figure 4-9: **First step**

comp-arc-list (Figure 4-11).

One final note is that when Q-D copying recurses into the arc-e of dg2 by following the temporary forwarding links while making a copy of dg1, the top node of dg2 will not be copied twice. This is so because when the top-level unify1 returns, the temporary forwarding from the top node of dg2 to dg1 is made, therefore, when the cyclic arc-e tries to make a copy of the top node of dg2, it finds that the top node is already forwarded to the top node of dg1. Since the top node of dg1 was already copied at the beginning of the Q-D copying of dg1, the already-made copy is simply returned (see the first **IF** in the Q-D Copying algorithm).

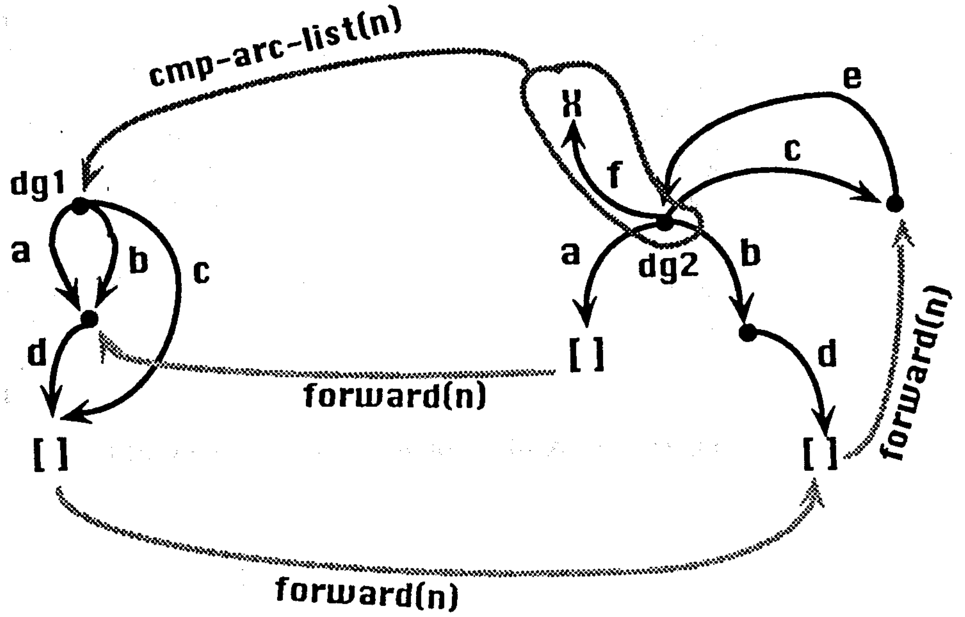


Figure 4-10: **At the end of time n**

Finally, we would like to provide the example of another cyclic graph unification (Figure 4-12), one which we already saw in Chapter 2 (Figure 2-2). It is the unification of the cyclic graphs which Pollard and Sag once regarded as not unifiable. We already claimed that from our definition of the subsumption relation, the unification of these graphs should be perfectly reasonable.

We promised in Chapter 1 and 2 that we would provide an algorithm that supports our

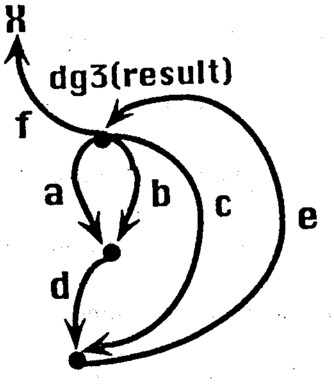


Figure 4-11: **and the result**

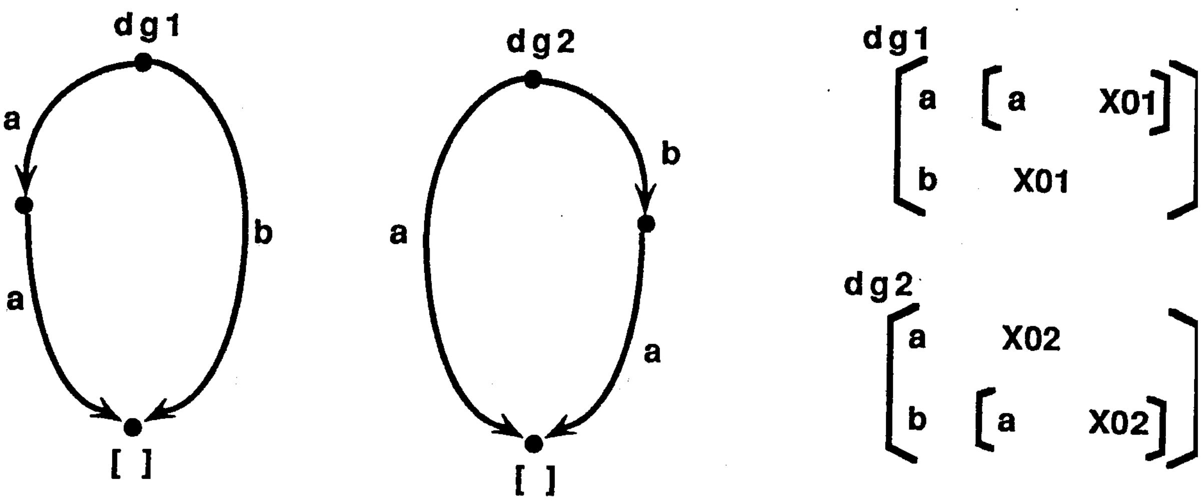


Figure 4-12: **Unification of another cyclic feature structure (Figure 2-2)**

definition of the subsumption and extention relations regardless of the existence of cycles. Here, we can see easily that the Q-D algorithm fulfills the promise. Actually, you will probably see that unification of these sturctures is rather trivial if you follow the steps of Q-D unification by hand. First we do unify1(dg1/$< a >$,dg2/$< a >$). Since dg2/$< a >$ is [] we forward from it to dg1/$< a >$ as in Figure 4-13. Next we do unify1(dg1/$< b >$,dg2/$< b >$) and this time dg1/$< b >$ is [] and we forward it to dg2/$< b >$ such that the result is as in Figure 4-14.

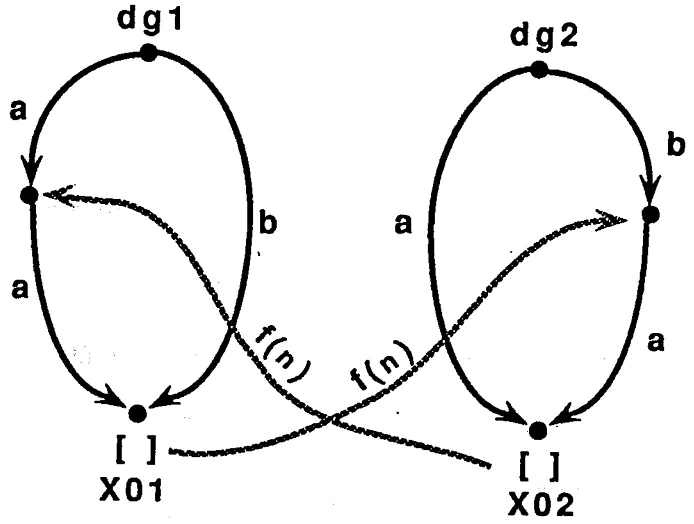
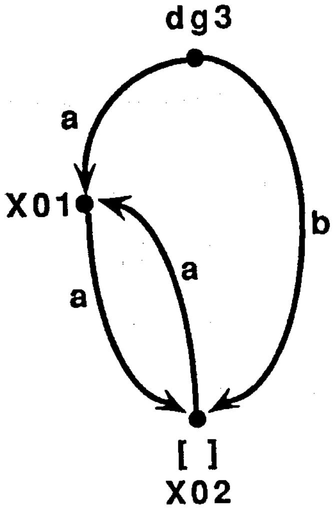


Figure 4-13: **Putting temporary forwarding links**

Finally, we esimply make a copy of dg1 as usual.[21] When we Q-D copy dg1/$< a, a >$ since it is forwarded to dg2/$< b >$ we copy the arc below which points to dg2/$< b, a >$. Since it is forwarded to dg1/$< a >$, we can copy that node. Now we find that this node was already copied when we traverse down the arc-a on dg1, so we simply return the copy that is already stored in the 'copy' field of dg1/$< a >$. This way, we can see that unification of these feature structures is possible and actually trivial using the Q-D scheme.



dg3
$$\begin{bmatrix} a & X01 & \begin{bmatrix} a & X02 \end{bmatrix} \\ b & X02 & \begin{bmatrix} a & X01 \end{bmatrix} \end{bmatrix}$$

Figure 4-14: and the result

## 4.3 Discussion

Incremental copying has been accepted as an effective method of minimizing over copying and eliminating early copying, as defined by Wroblewski. However, while being effective in minimizing over copying (it over copies only in some special cases of convergent arcs), incremental copying is ineffective in eliminating early copying as we define it.[22]

[21] Recall that in our notation, the values with the same tagging X01 in the distinct feature structures do not indicate the same value. the taggings Xmn are simply put in order of appearance from the root nodes within one graph. Therefore, the node X01 of dg1 and node X01 of dg3 are distinct nodes. Actually, as a matter of correspondence in this figure, X02 of dg3 corresponds to X01 of dg1.

[22] 'Early copying' will henceforth be used to refer to early copying as defined by us.

Incremental copying is ineffective in eliminating early copying for two reasons. First, when incremental copying unification is performed, any copies created up to the point of failure in the same subgraph of a shared arc will be wasted, as seen in Figure 4-15.

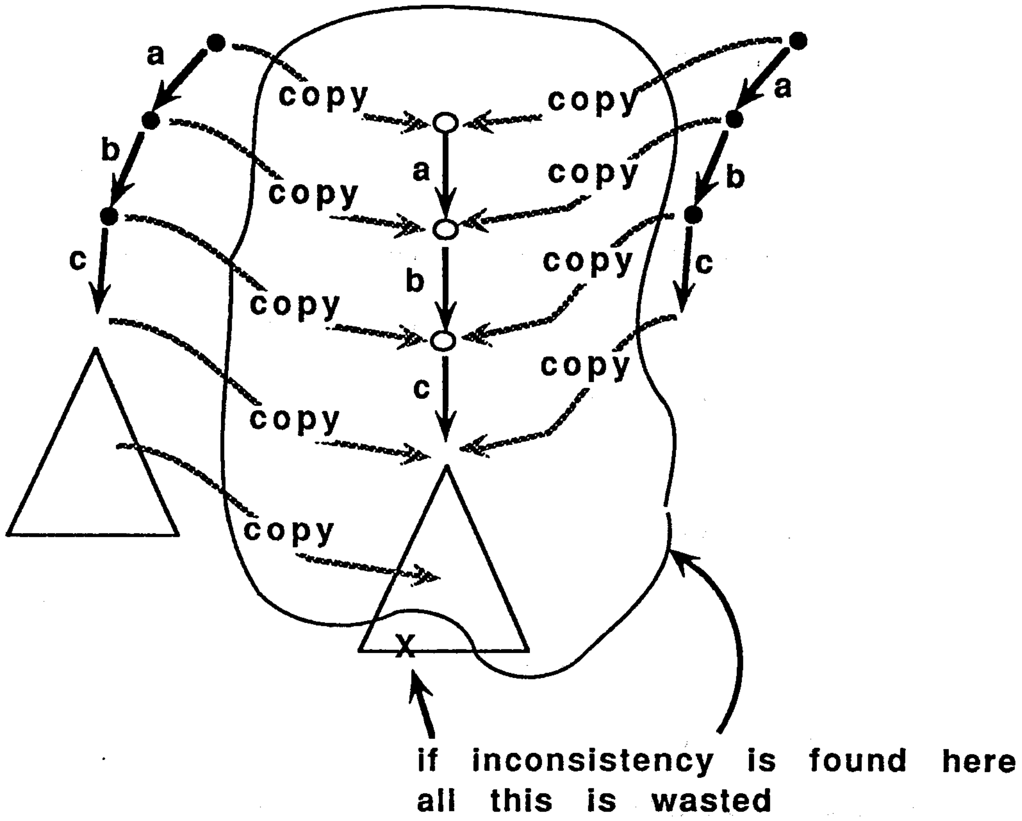
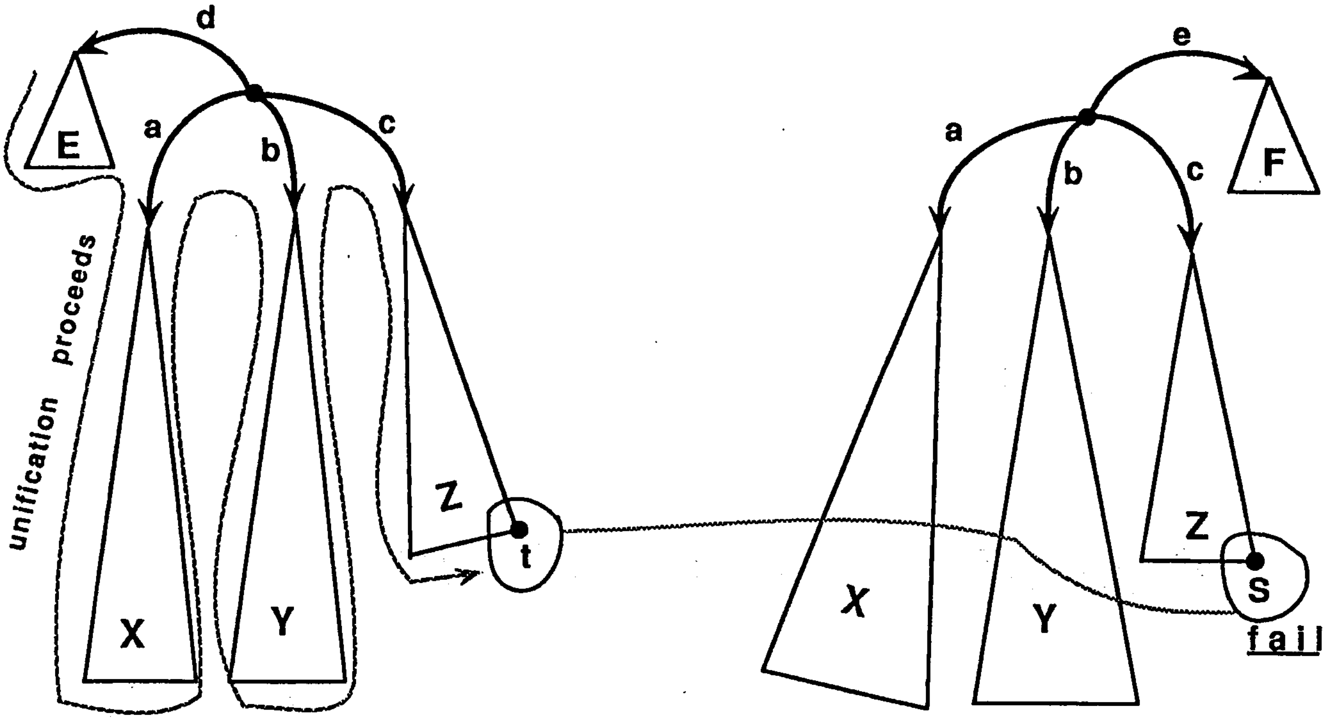


Figure 4-15: Early copy of incremental scheme within the same subgraph

Second and more significantly that since the recursive calls into the shared arcs are non-deterministic (independent of each other), there is no way for one particular recursion into a shared arc to know the result of future recursions into other shared arcs. Therefore, even if a particular recursion into one arc succeeds (with minimum over copying and no early copying in Wroblewski's sense), other arcs may eventually fail; thus the copies that are created in the successful arcs are all wasted. Figure 4-16 shows such an example. If incremental unification proceeds unifying the subgraphs E,X,Y, and then Z. At some deep position of Z, if unification failure is found, not only are nodes in Z wasted (as we saw in Figure 4-15) but all of the copying created in E,X,Y will also be wasted. By structure-sharing of unmodified graphs, Kogure's and Emele's schemes can avoid wasting the subgraph E (i.e., a complement graph), but their scheme cannot avoid wasting X and Y (Figure 4-16). Note that this is inherent and unavoidable in incremental schemes, since by definition, these schemes must produce copies as they proceed. Since each recursive calls to shared arcs are non-deterministic, future event in other recursive calls are not predictable. In order to avoid this problem, incremental schemes will have to delay all copying until after entire top-level unification. This will mean that these unifications will no longer be incremental. Thus, fully delaying copying in incremental schemes to avoid early copying would make their control structures essentially no different from Q-D and reversible (Karttunen) schemes. In other words, we can also view the Q-D scheme as a fully lazy scheme without overhead for delaying.

The difference between the Q-D scheme and the incremental scheme becomes apparent when the used grammar is sufficiently large, containing large subgraphs which may be over-copied by the incremental scheme. As we will see in the data in Chapter 6, by avoiding the Early Copying, the proposed algorithm runs at about twice the speed of Wroblewski's[1987] algorithm. The control structure of our algorithm is identical to that of Pereira[1985]. However, in Pereira's method, a result graph is represented as a combination of the original graph ('skeleton') and the updates (new arcs to be added to create the result stored in 'environment'). Thus the result graph is dynamically created whenever it is needed. This causes the log(d) overhead (where d is the number of nodes in a graph) to assemble the whole graph everytime the node is accessed. In the proposed scheme, instead of storing changes to the argument graphs in the environment, we store the changes in the graph structure themselves (non-destructively); therefore, there will be no overhead associated with node accesses. We share the principle of storing changes in a restorable way with Karttunen's[1986] reversible unification and copy graphs only after a



**Suppose X, Y were successful but unification failure was found somewhere in Z; then X, Y and Z until detection of failure are wasted. Kogure and Emele avoid copying of E but X, Y, Z will be copied and these copies are all wasted.**

Figure 4-16: **Unavoidable massive early copying of incremental schemes**

successful unification. In the Karttunen's method, whenever a destructive change is about to be made, the attribute value pairs[23] stored in the body of the node are saved into an array. These values are restored after the top level unification is completed. (A copy is made prior to the restoration operation if the unification was a successful one.) Thus, in Karttunen's method, each node in the entire argument graph that has been destructively modified must be restored separately by retrieving the attribute values saved in an array and by resetting the values into the dag structure skeletons. In the Q-D method, one increment to the global counter can invalidate all the changes made to the nodes.

[23] I.e., arc structures: 'label' and 'value' pairs in our vocabulary.

# Chapter 5

# Quasi-Destructive Graph Unification with Structure-Sharing

## 5.1 Introduction

In the previous chapter, we presented the following observation about graph unification:

> **Unification does not always succeed, and**
>
> **Copying is an expensive operation.**

We proposed the following two principles for fast graph unification based upon the above observations:

- **Copying should be performed only for successful unifications.**
- **Unification failures should be found as soon as possible.**

Thus, we eliminated Over Copying and Early Copying (as defined in the previous chapter). In this chapter, we propose another design principle for graph unification based upon yet another observation :

**Unmodified subgraphs can be shared.**

At least two schemes (namely [Kogure, 1990] and [Emele, 1991]) have been proposed recently based upon this observation; however, both schemes are based upon the incremental copying scheme. As described in previous chapter, incremental copying schemes inherently suffer from *Early Copying,* as defined in this thesis. This is because, when a unification fails, the copies that were created up to the point of failure are wasted if copies are created incrementally. By way of definition we would like to categorize the sharing of structures in graphs into Feature-Structure Sharing (FS-Sharing) and Data-Structure Sharing (DS-Sharing). Below are our definitions:

- **Feature-Structure Sharing:** Two or more distinct paths within a graph share the same subgraph by converging on the same node – equivalent to the notion of *structure sharing* or *reentrancy* in linguistic theories (such as in [Pollard and Sag, 1987]).
- **Data-Structure Sharing:** Two or more distinct graphs share the same subgraph by converging on the same node – the notion of *structure-sharing* at the data structure level. [Kogure, 1990] calls copying of such structures *Redundant Copying.*

Virtually all graph-unification algorithms support FS-Sharing and some support DS-Sharing with varying levels of overhead. In this chapter, we propose a scheme of graph unification based upon a quasi-destructive graph unification method that attains DS-Sharing with virtually no overhead for structure-sharing. Henceforth, in this thesis, structure-sharing refers to DS-sharing unless otherwise noted. We will see that the introduction of structure-sharing to quasi-destructive unification attains another two-fold increase in run-time speed. The graphs handled in the scheme can be any directed graph and cyclicity is handled without any algorithmic additions.

Our design principles for achieving structure-sharing in the quasi-destructive scheme are:

- **Atomic and Top nodes can be shared**[1] : Atomic nodes can be shared safely since they never change their values. Top nodes can be shared[2] since Top nodes are always forwarded to some other nodes when they unify.
- **Complex nodes can be shared unless they are modified** : Complex nodes can be considered modified if they are a target of the forwarding operation or if they received the current addition of complement arcs (into comp-arc-list in a quasi-destructive scheme).

By designing an algorithm based upon these principles for structure-sharing while retaining the quasi-destructive nature of our algorithm, our scheme eliminates Redundant Copying while eliminating both Early Copying and Over Copying.

Figure 5-1 shows how structure sharing in the proposed scheme will be attained. All the subgraphs which are not modified are shared by the result graph. In the subgraph where modification occurred, only the path above the modified node is copied and the nodes in the path below the modified node are simply shared with the original graphs. In the next section, we will see how this can be done in the Q-D scheme.

## 5.2 Quasi-Destructive Graph Unification with Structure-Sharing

In order to attain structure-sharing during Quasi-Destructive graph unification, no modification is necessary for the unification functions described in the previous section. This section describes the quasi-destructive copying with structurȩ-sharing (QDSS) which replaces the original copying algorithm. Since unification functions are unmodified, the Q-D unification without structure-sharing can be mixed trivially with the Q-D unification with structure-sharing if such a mixture

[1] Recall that atomic nodes are nodes that represent atomic values. Top nodes are nodes that represent variables.

[2] As long as the unification operation is the only operation to modify graphs.

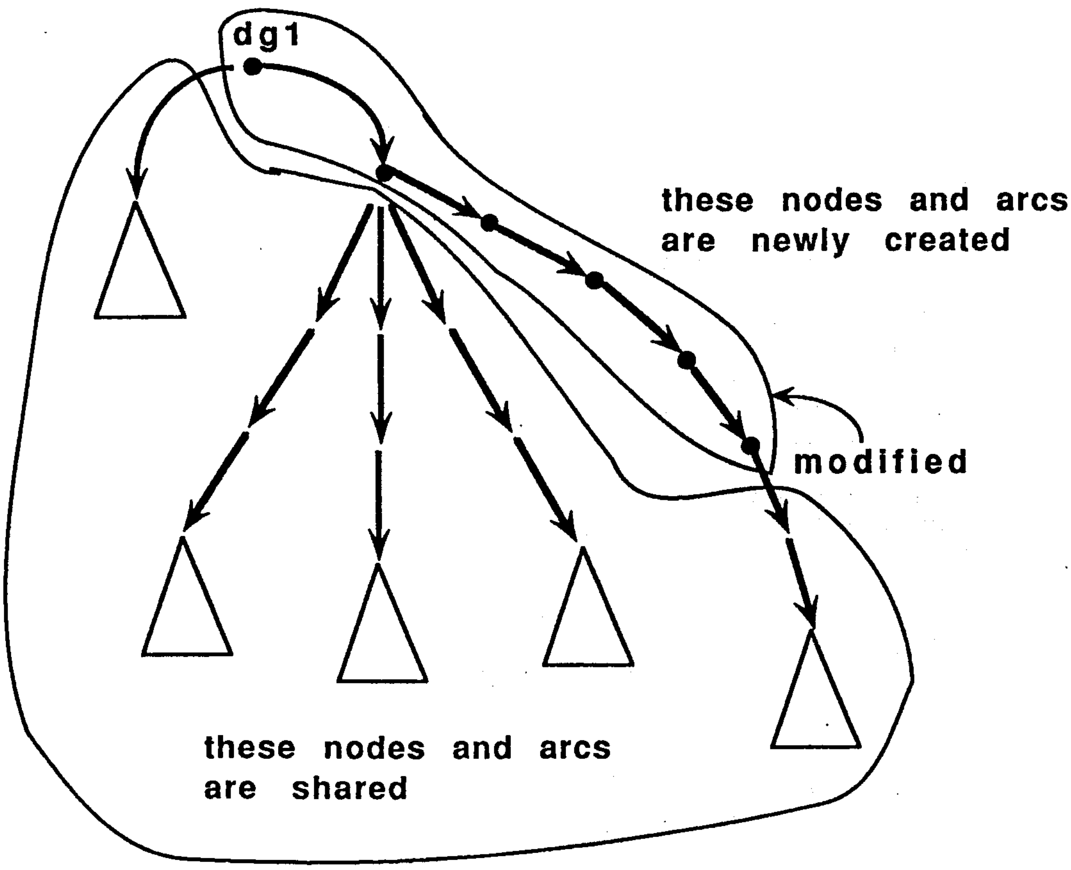

dg1
these nodes and arcs are newly created
modified
these nodes and arcs are shared

Figure 5-1: **How Redundant Copying is Eliminated**

is desired (by simply choosing different copying functions). Informally, the Q-D copying with structure-sharing is performed in the following way. Atomic and Top nodes are shared. A complex node is shared if no nodes below that node are changed (a node is considered changed by being a target of forwarding or having a valid comp-arc-list). If a node is changed then that information is passed up the graph path using a multiple-value binding facility when a copy of the node is recursively returned. Two values are returned, the first value being the copy (or original) node and the second value being the flag indicating whether any of the nodes below that node (including that nodes) have been changed. Atomic and Top nodes are always shared; however, they are considered changed if they were targets of forwarding, such that the 'changed' information is passed up. If the complex node is a target of forwarding, and if no node below that node is changed, then the original complex node is shared; however, the 'changed' information is passed up when the recursion returns. Below is the actual algorithm description for Q-D copying with structure-sharing.

Q-D COPYING WITH STRUCTURE-SHARING

```
FUNCTION copy-dg-with-comp-arcs-share(dg-underef);
  dg ← dereference-dg(dg-underef);
  IF (dg.copy is non-empty AND
      dg.copy.generation = *unify-global-counter*) THEN
      values(dg.copy, :changed);[3]
  ELSE IF (dg = dg-underef) THEN
      copy-node-comp-not-forwarded(dg);
  ELSE copy-node-comp-forwarded(dg);
END;

FUNCTION copy-node-comp-not-forwarded(dg);
  IF (dg.type = :atomic) THEN values(dg,nil);
  ;; return original dg with 'no change' flag.
  ELSE IF (dg.type = :Top) THEN values(dg,nil);
  ELSE
     IF (dg.comp-arc-list is non-empty AND
         dg.generation = *unify-global-counter*) THEN
         newcopy ← create-node();
         newcopy.type ← :complex;
         newcopy.generation ← *unify-global-counter*;
         dg.copy ← newcopy;
         FOR ALL arc IN dg.arc-list DO
            newarc
              ← first value of copy-arc-and-comp-arc-share(arc);
            push newarc into newcopy.arc-list;
         FOR ALL comp-arc IN dg.comp-arc-list DO
            newarc
              ← first value of copy-arc-and-comp-arc-share(comp-arc);
            push newarc into newcopy.arc-list;
         dg.comp-arc-list ← nil;
         values(newcopy,:changed);
     ELSE
         state ← nil, arcs ← nil;
         dg.copy ← dg[4], dg.generation ← *unify-global-counter*;
         FOR ALL arc IN dg.arc-list DO
            newarc,changed ← copy-arc-and-comp-arc-share(arc); [5]
            push newarc into arcs;
            IF (changed has value) THEN
                state ← changed;
         IF (state has value) THEN
             newcopy ← create-node();
             newcopy.type ← :complex;
             newcopy.generation ← *unify-global-counter*;
             newcopy.arc-list ← arcs;
             dg.copy ← newcopy;
             values(newcopy,:changed);
         ELSE dg.copy ← nil; ;;reset copy field
               values(dg,nil);
END;
```

```
FUNCTION copy-node-comp-forwarded(dg);
  IF (dg.type = :atomic) THEN values(dg,:changed);
  ;; return original dg with 'changed' flag.
  ELSE IF (dg.type = :Top) THEN values(dg,:changed);
  ELSE
     IF (dg.comp-arc-list is non-empty AND
         dg.generation = *unify-global-counter*) THEN
         newcopy ← create-node();
         newcopy.type ← :complex;
         newcopy.generation ← *unify-global-counter*;
         dg.copy ← newcopy;
         FOR ALL arc IN dg.arc-list DO
            newarc
              ← first value of copy-arc-and-comp-arc-share(arc);
            push newarc into newcopy.arc-list;
         FOR ALL comp-arc IN dg.comp-arc-list DO
            newarc
              ← first value of
                 copy-arc-and-comp-arc-share(comp-arc);
            push newarc into newcopy.arc-list;
         dg.comp-arc-list ← nil;
         values(newcopy,:changed);
     ELSE
         state ← nil, arcs ← nil;
         dg.copy ← dg, dg.generation ← *unify-global-counter*;
         FOR ALL arc IN dg.arc-list DO
            newarc,changed ← copy-arc-and-comp-arc-share(arc);
            push newarc into arcs;
            IF (changed has value) THEN
                state ← changed;
         IF (state has value) THEN
            newcopy ← create-node();
            newcopy.type ← :complex;
            newcopy.generation ← *unify-global-counter*;
            newcopy.arc-list ← arcs;
            dg.copy ← newcopy;
            values(newcopy,:changed);
         ELSE dg.copy ← nil;
               values(dg,changed); ;; considered changed
END;

FUNCTION copy-arc-and-comp-arc-share(input-arc);
  destination,changed
        ← copy-dg-with-comp-arcs-share(input-arc.value);
  IF (changed has value) THEN
     label ← input-arc.label;
     value ← destination;
     values(a new arc with label and value,:changed);
  ELSE values(input-arc,nil); ;; return original arc
```

**END;**

Let us review a few examples. Figure 5-2 represents a unification between two graphs, each containing large subgraphs (shown as triangles).

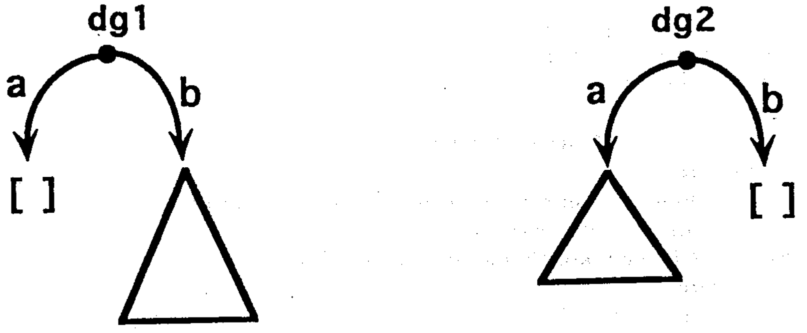


Figure 5-2: **A simple input example with large subgraphs**

The Q-D algorithm without structure sharing described in the previous chapter would copy these subgraphs. However, with the introduction of the structure-sharing scheme, only one node (i.e, the top node of the result graph dg3) and only two arcs (arc-a and arc-b) are copied (Figure 5-3). The subgraphs dgl/$< b >$ and dg2/$< a >$ are not copied at all since there was no modification in these subgraphs. Therefore, the original top nodes of the subgraphs are directly pointed to by the newly created arcs arc-a and arc-b for dg3. The arcs arc-a and arc-b are copied since the top nodes of the subgraphs were targets of forwarding; therefore, :changed information is passed up. In our algorithm, if a complex node is a target of forwarding although no copies are made of that node; it is considered modified and the arcs and nodes above that node are copied. Therefore, there will be one copy node, the top node of dg3, and two new arcs pointing to the original nodes created in this unification.

We would like to provide one more example of Q-D structure-sharing (QDS). This one is

[3] 'Values' return multiple values from a function. In our algorithm, two values are returned. The first value is the result of copying, and the second value is a flag indicating if there was any modification to the node or to any of its descendants.

[4] Temporarily set copy of the dg to be itself.

[5] Multiple-value-bind call. The first value is bound to 'newarc', and the second value is bound to 'changed'.

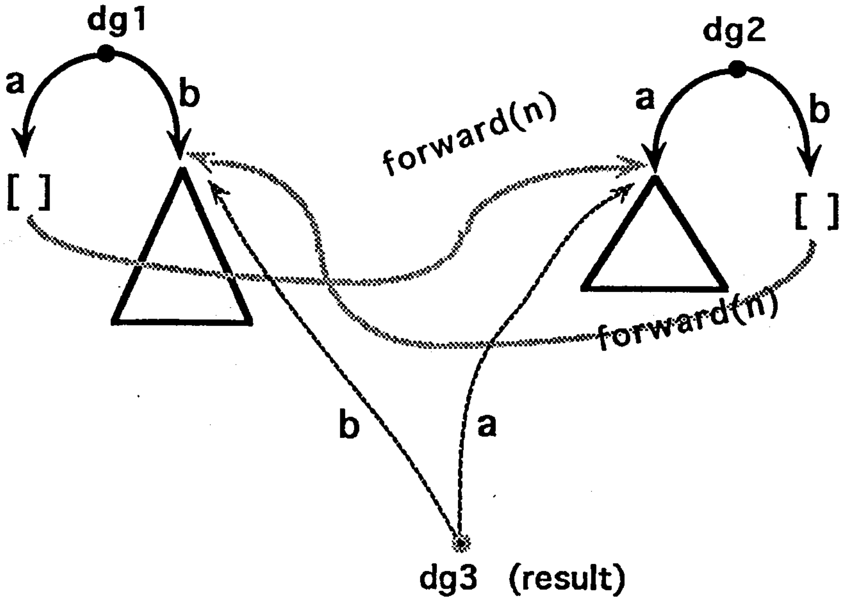


Figure 5-3: **The result of Q-D with Structure-sharing (QDS)**

bit more complicated (Figure 5-4).

First, Q-D unification is performed on the input graphs, as we already learned in the previous chapter putting temporary forwarding links and a comp-arc-list (Figure 5-5).

Now, after Q-D unification we do the QDS Copying (Figure 5-6). The top node dg3 will be created since changes below will return the :changed flag upwards when recursive unification returns. The arc-e is not copied at all since there was no change and therefore, the entire arc-e is simply put in arc-list of dg3. dg1/$< a >$ is not copied but the arc a of dg1 is copied since it is the target of forwarding. By the same token dg2/$< b >$ is not copied but the arc-b of dg2 is copied. This way, only one node (i.e., the root of dg3) and two arcs (a and b) are created to produce the result graph as seen in the figure.

The following two figures (Figures 5-7, 5-8) show the similar structure-sharing. First temporary forwarding is performed after successful unifications (Figure 5-7). Then this time because

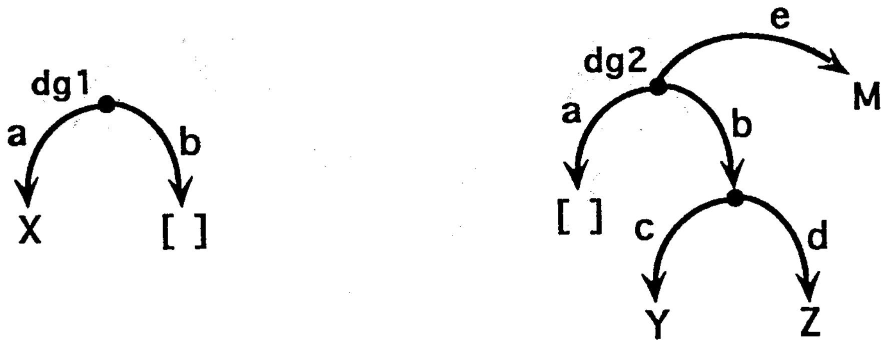


Figure 5-4: **Another example**

dg1/< $a, b$ > was a target of forwarding, although dg1/< $a, b$ > is not copied the arcs and nodes leading to that node is copied. Thus, there will be 2 new nodes and 3 new arcs with this example (Figure 5-8).

## 5.3 Discussion

The structure-sharing scheme introduced in this section made the Q-D algorithm run significantly faster. Provided in Chapter 6, the structure-sharing version of the Q-D algorithm (called QDS or QDSS) runs at more than twice the speed of the non-structure sharing version (QD). It runs at about 4 times the speed of Wroblewski's algorithm. The source of the gain is apparent in that the number of created copies (nodes) and arcs is significantly reduced in the QDS scheme. We will see in Chapter 6 that whereas the QD scheme created about 75 percent of copies created by Wroblewski's algorithm; the QDS scheme only created 19 percent

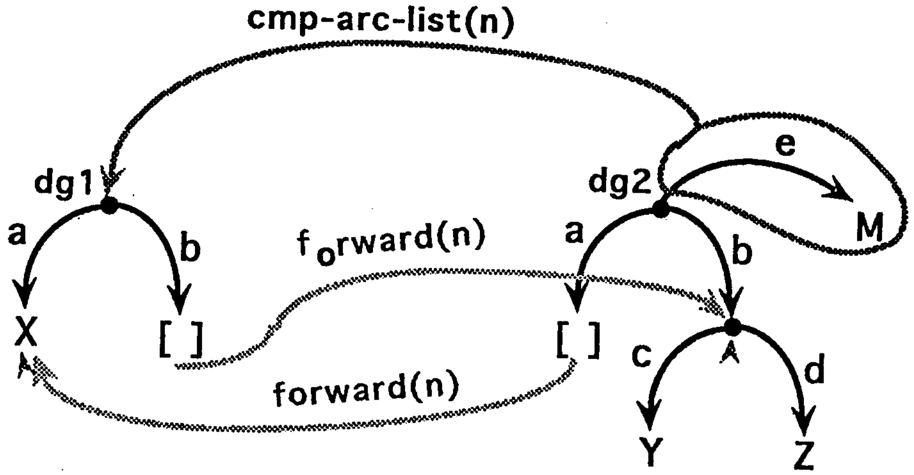


Figure 5-5: **at the end of time n**

of Wroblewski's algorithm. The original gain of the Q-D algorithm was due to the fact that it does not create any Over Copies or Early Copies, whereas the incremental copying scheme inherently produces Early Copies when a unification fails. The proposed scheme makes the Q-D algorithm completely avoid Redundant Copies as well by copying only the lowest nodes that need to be copied due to destructive changes caused by only the successful unifications. Since there will be no overhead associated with structure-sharing (except for passing up two values, i.e., ':changed/nil' and 'the result node', instead of one (result node) when recursion for copying returns), the introduction of structure-sharing to the Q-D scheme is an ideal addition to the algorithm.

Pereira ([Pereira, 1985]) attained structure-sharing by having the result graph share information with the original graphs by storing changes to the 'environment'. As discussed in the previous section, there will be the log(d) overhead (where d is the number of nodes in a graph)

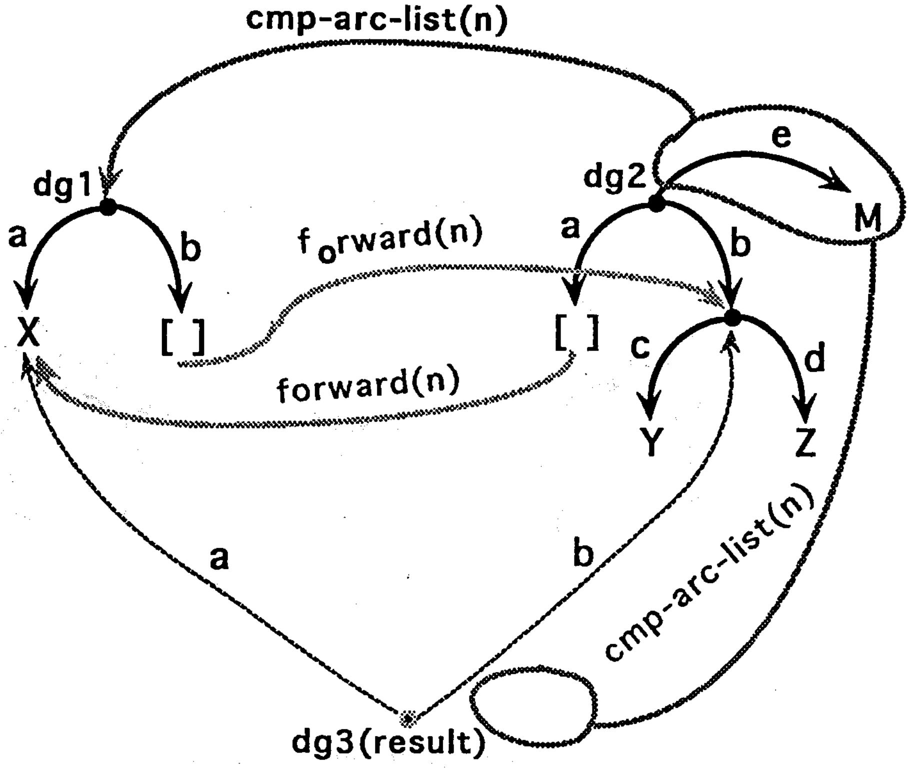


Figure 5-6: **and the result**

associated with Pereira's method that is required during node access in order to assemble the whole graph from the 'skeleton' and the updates in the 'environment'. In the proposed scheme, since the arcs directly point to the original graph structures, there will be no overhead for node accesses. Also, during unification, since changes are stored directly in the nodes (in the quasi-destructive manner) there will be no overhead for reflecting the changes to graphs during unification.

We share the principle of storing changes in a restorable way with Karttunen's[1986] reversible unification and copy graphs only after a successful unification. Although both overcopying and early-copying are avoided in his scheme, redundant-copying is not solved. This is so because there is no structure-sharing involved in his scheme and therefore, the subgraphs in the input graphs which are never modified are still copied after successful unifications. As discussed

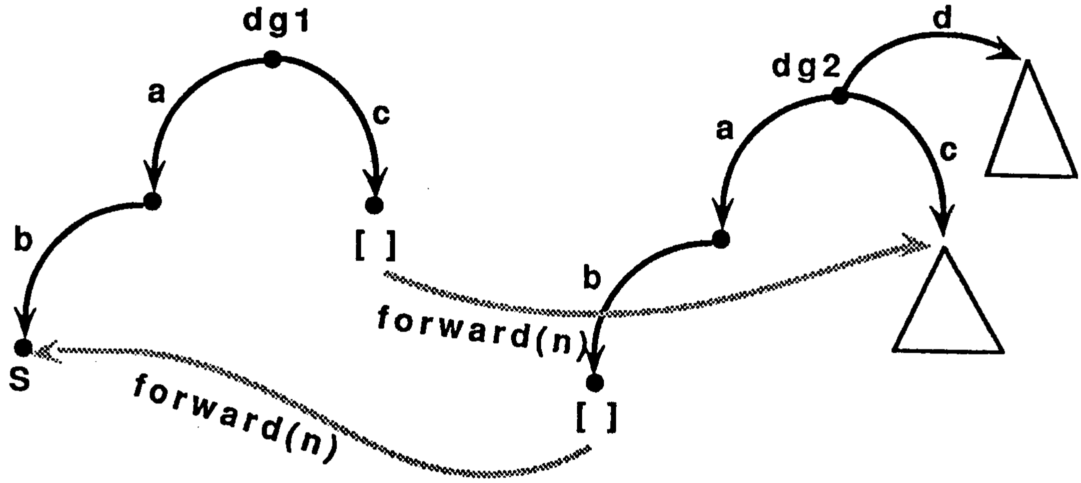


Figure 5-7: **One more example**

in Chapter 4, although Emele and Kogure introduced structure-sharing, because their central control structure is inherently incremental, early copying is inevitable if failure is detected as we saw in Figures 4-15 and 4-16.

There is one potential problem with the structure-sharing idea which is shared by each of the schemes, including the method proposed here. This happens when operations other than unification modify the graphs. (This is typical when a parser cuts off part of a graph for subsequent analysis[6].) When such operations are performed, structure-sharing of Top (variable) nodes may cause problems when a subgraph containing a Top is shared by two different graphs and when these graphs are used as arguments of a unification function (either as part of the same input graph or as elements of dg1 and dg2). When a graph that shares a Top node is not used in its entirety, then the represented constraint postulated by the path leading to the Top node is no longer the same. Therefore, when such a graph appears in the same unification along with some other graph with which it DS-Shares the same Top node, there will be a false

[6] For example, many parsers cut off a subgraph of the path X0 for applying further rules when a rule is accepted.

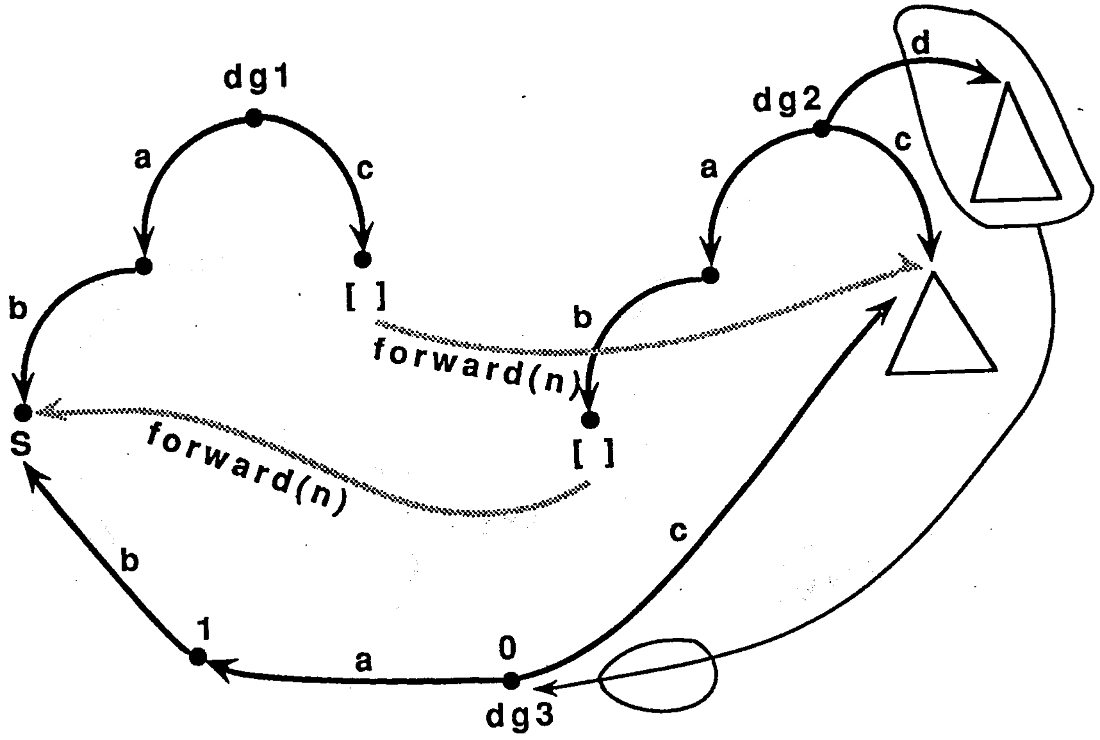


Figure 5-8: The result

FS-Sharing. (If the graph is used in its entirety this is not a problem since the two graph paths would unify anyway.) This problem happens only when neither of the two graphs that DS-Shares the same Top node was unified against some other graph before appearing in the same unification.[7] (If either was once unified, forwarding would have avoided this problem).

Consider the figures below. Unifying the shared graphs dg1 and dg2 are fine with these two examples in Figure 5-9.

Also, the following correctly fails (Figure 5-10).

But the unification in Figure 5-11 incorrectly fails.

The question here is whether the cases such as Figure 5-11 is possible during parsing at all. The answer seems that as long as the unification operation is the only operation on graphs

[7] Such cases may happen when the same rule (such as V ⇒ V) augmented with a heavy use of convergence in the Top nodes is applied many times during a parse.

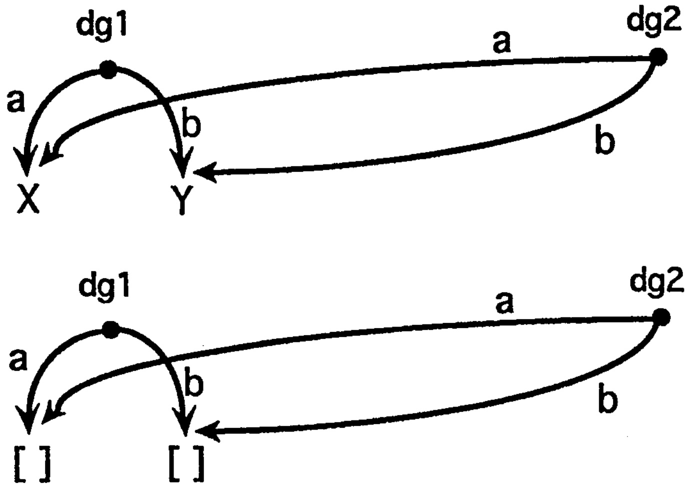


Figure 5-9: **Unifying shared graphs – correct results**

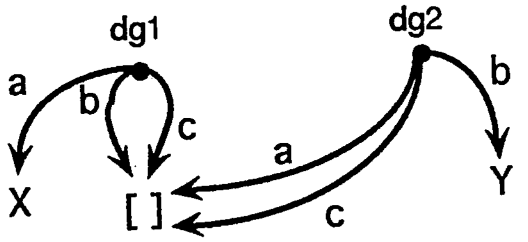


Figure 5-10: **Unifying shared graphs – still correct result**

then the case like Figure 5-11 is not possible. It is because unification only adds information and never subtracts information. Because of the monotonic increase nature of informational content in unification, if two graphs share a subgraph (or a node), then the two graphs must have at least one path (arc) that they share the labels. This is so because structure-sharing is performed during QDS Copying between the original graph and the result graph and therefore, if two distinct graphs share a same node then the shared node (subgraph) comes from the same original graph. Therefore, the path leading to the node must be identical in two graphs.

However, if a parser modifies the input graphs destructively by deleting arcs from the graphs

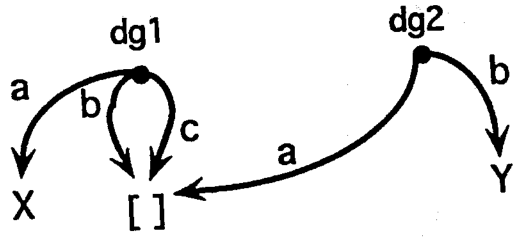


Figure 5-11: **incorrect to fail**

then the case like Figure 5-11 may happen. More figures to follow illustrates this problem:

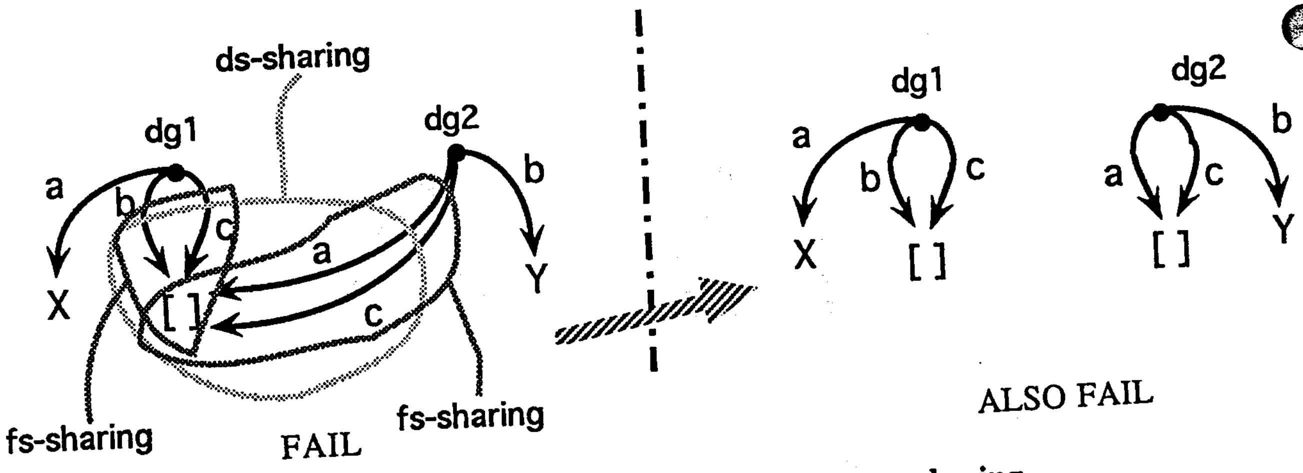


Figure 5-12: **Correct result by structure-sharing**

Figure 5-12 illustrates the corresponding input graphs of Figure 10 without structure-sharing. It is easy to see that the result is correct with structure-sharing.

However, if we delete arc-c from dg2 in Figure 5-12 (5-10) as shown in Figure 5-13, incorrect result will be produced. This way, cutting off a part of a graph is a dangerous operation when structure-sharing of variables is introduced. Many parsers cut off subgraph of X0 paths and pass up the subgraph to become subgraph of Xn in other rule graphs. X0 (mother) subgraph of the current unification are used as Xn (daughter) subgraph of future unifications. We now see that such a scheme can cause a problem. The better method for passing up the mother

But, if we delete the arc-c from dg2:

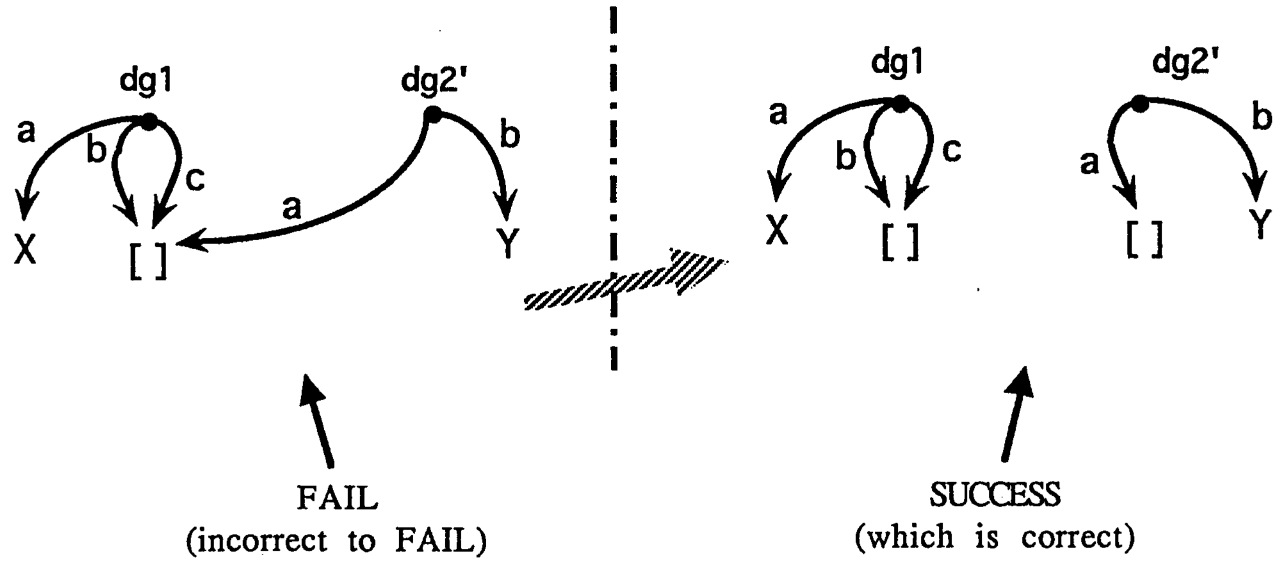


Figure 5-13: **Incorrect result after deleting an arc**

information using a unification-based grammar is simply to represent the root (mother) by <> instead of $< X0 >$ as was used in PATR-II like formalisms. The grammar we used for our experiments was written that way and the parser does not cut of the subgraph of $< X0 >$; instead it passes the whole root node upward through the X-bar levels. Even if this problem is solved there is another similar problem. It is due to the fact that reentrant variables extend non-reentrant variables. Therefore, if a reentrant path containing a variable is unified with a non-reentrant path with the same features, than the resulting reentrant graph (if the reentrant one was dg1) would share the variable with the non-reentrant one. If this happens and if the original non-reentrant graph and result reeentrant graph was within the same parse, then again incorrect FS-Sharing may result. Since, after a whole sentential parse, the constituent built during the parse is no longer used, this problem would arise only during one parse.

As describe above, care should be taken in treating structure-sharing of variables. The methods to avoid such a problem include the following: 1) As long as convergence of Top nodes is used for features that are not passed up during parsing, the problems do not affect the result of parse in any way – which is the case with the grammar in our experiment; and 2) Whenever the same rule graph is used twice within a parse, make a copy of the rule graph when it is used for the second time – which is the method taken in ATR's ASURA system ([Takahashi, *et al*, 1992]). Although, structure-sharing of variables needs extra care, the efficiency gain from sharing variables should more than offset the efforts that need to be taken in order to guarantee the correct behaviour. The chart in Appendix VI is taken from [Takahashi, *et al*, 1992]. It is data taken from the above mentioned ASURA experiments by Takahashi, *et al* using ATR's latest large-scale grammar adopting the Q-D algorithms. The data shows that if structure sharing of variables were not performed, there would be only a 40 percent reduction in the number of nodes copied, compared to the non-structure-sharing Q-D algorithm. However, if bottom nodes are shared, there is an 85 percent reduction from the non-structure-sharing Q-D scheme. Since its current implementation copies a rule graph the second time it is using within one parse, if we can avoid the second copying of rules as well, there could be an even greater reduction in the amount of copying performed.

# Chapter 6

# Empirical Results

## 6.1 Comparison using actual grammar

This section describes the empirical results obtained from our sample implemtations of the Q-D and QDS algorithms. Table 6.1 shows the summary of the results of our experiments using an HPSG-based Japanese grammar for the conference registration telephone dialogue domain. We used 16 sample sentences which are provided in the Appendix I. A representative portion of the grammar is shown in Appendix II. The grammar used in the experiments was originally developed by ATR as a large scale spoken-Japanese language grammar (containing over 10,000 grammar nodes) and is scaled down (about 3,000 grammar nodes) to run on a Sun Sparc2 with 28 mega bytes of physical memory at CMU.

We used Earley's parsing algorithm for the experiment. Although it is scaled down from the ATR's current ASURA grammar, it covers many of the important linguistic phenomena in spoken Japanese. The covered phenomena include coordination, case adjunction, adjuncts, control, slash categories, zero-pronouns, interrogatives, WH constructs, and some pragmatics (speaker, hearer relations, politeness, etc.) ([Yoshimoto and Kogure, 1989]). The 16 sample telephone conversation sentences used in the experiments varied from short sentences (one word, i.e., *hai* 'yes') to relatively long ones (such as *soredehakochirakarasochiranitourokuyoushiwooookuriitashimasu*, which means 'In that case, we [speaker] will send [polite] you [hearer] (the) registration form.'). Thus, the number of (top-level) unifications per sentence varied widely (from 7 to over 4,000).

'Unifs' in the table represents the total number of top-level unifications during a parse (i.e, the number of calls to the top-level 'unify-dg', and not 'unify1'). Thus, 'Unifs' is not the total number of unifications recursively called during a parse. It is only the number of top-level unifications called by the parser during the analysis of a sentence. Normally, during a parse, 'unify1' is called several times more often than 'unify-dg' ('unifys'). For example, with QDS, for the parse of the sentence 12, 'unify-dg' ('Unifs') were called 3,421 times and during this parse, unify1 was called 22,674 times. For sentence 13, it was 4,274 and 27,605 respectively. 'USrate' represents the ratio of successful unifications to the total number of unifications. 'Number of Copies' represents the number of nodes created during each parse. 'CPU-Time (non-gc msec user)' is the actual parsing time for a sentence in milli-seconds (1/1000th of a second) not counting the time taken for garbage collection. (The parser and the unification algorithms are implemented in CommonLisp). 'CPU-Time (total msec user)' includes the time required for garbage collection that proceeds in the background.

The algorithms compared were Quasi-Destructive Graph Unification with Structure-Sharing (QDS), Quasi-Destructive Graph Unification (Q-D), Wroblewski's algorithm (W), and Karttunen's algorithm (K). These algorithms are described in Chapter 5 (QDS), Chapter 4 (Q-D), and Chapter 3 (Karttunen[1] and Wroblewski[2]) of this thesis. We did not adopt Pereira's algo-

[1] We updated his algorithm slightly to handle cycles and also only one array is used in our implementation to store the contents of original graphs.

[2] We updated his algorithm as well to handle cycles.

rithm for the experiments since Karttunen's algorithm has been reported to be more efficient. Also, we could not find an efficient way of handling cycles using Pereira's algorithm. We adopted Wroblewski's algorithm (enhanced by Kogure's method for handling cycles[3]) as representative of the incremental copying schemes since a significant speed-up over Wroblewski's has not been reported in incremental schemes. Additionally, we could not find a method to handle cycles using Emele's algorithm efficiently. Because of the ease of implementing Wroblewski's algorithm, it should be easy to compare the performance of any future incremental schemes against the performance of Wroblewski's algorithm and to indirectly compare them with the performance of Q-D, QDS and Karttunen's algorithms reported in this thesis. The Earley parser and the unification algorithms are written in CommonLisp[4] and are run on a SUN Sparc2 with 28 mega bytes of RAM.

Using the data shown in the Table 6-1, Figure 6-1 plots the relation between the number of nodes created (i.e., number of copies created) during a parse ('Num of Copies') and the number of unifications during a parse ('Unifs') for the sample 16 sentences. We can see that the increase in the number of nodes created during a parse is approximately linear to the increase in the number of unifications during a parse consistently for the 16 sentences. The amount of copies stays at around 75 percent of Wroblewski's algorithm using the Q-D and Karttunen algorithm. The Q-D and Karttunen algorithms behave in the same manner since they both create copies only after successful unifications and neither use structure-sharing. About half of the unifications were failures during the parses and the copies created during unifications until the detection of failures in Wroblewski's algorithm are the source of this reduction in

[3] Cycles can be handled in Wroblewski's algorithm by checking whether an arc with the same label already exists when arcs are added to a node. If such an arc already exists, we destructively unify the node which is the destination of the existing arc with the node which is the destination of the arc being added. If such an arc does not exist, we simply add it ([Kogure, 1989]). Thus, cycles can be handled very cheaply in Wroblewski's algorithm.

[4] Allegro CL 4.0.1 [SUN 4].

Comparison of four methods - Number of Copies and CPU user time

| Sent# | Unifs | USrate | Num of Copies | | | CPU-Time (non-gc msec user) | | | | CPU-Time (total msec user) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | QDS | Q-D&K | W | QDS | Q-D | W | K | QDS | Q-D | W | K |
| 1 | 7 | 0.42 | 18 | 79 | 96 | 233 | 184 | 250 | 250 | 233 | 184 | 250 | 250 |
| 2 | 370 | 0.38 | 1821 | 6333 | 8118 | 1867 | 1917 | 2534 | 9883 | 1867 | 1917 | 3900 | 12483 |
| 3 | 19 | 0.21 | 26 | 111 | 172 | 267 | 267 | 267 | 250 | 267 | 267 | 267 | 250 |
| 4 | 219 | 0.51 | 1263 | 4654 | 6036 | 1300 | 1567 | 1933 | 4334 | 1483 | 1567 | 1933 | 5850 |
| 5 | 2433 | 0.38 | 12321 | 50220 | 66204 | 11516 | 16233 | 24033 | 352817 | 18217 | 41050 | 51467 | 437567 |
| 6 | 245 | 0.37 | 937 | 3670 | 4569 | 1200 | 1450 | 1667 | 3850 | 1200 | 1450 | 1667 | 3850 |
| 7 | 7 | 0.42 | 18 | 79 | 96 | 233 | 200 | 300 | 250 | 233 | 200 | 300 | 250 |
| 8 | 314 | 0.48 | 1269 | 6009 | 7426 | 1600 | 2584 | 2800 | 7066 | 1600 | 2584 | 2800 | 10166 |
| 9 | 1996 | 0.32 | 8718 | 38024 | 53354 | 10334 | 10784 | 17516 | 223683 | 13067 | 30700 | 39383 | 320583 |
| 10 | 2811 | 0.42 | 13894 | 59762 | 86448 | 19617 | 28883 | 42849 | 596266 | 27067 | 54433 | 95733 | 784700 |
| 11 | 223 | 0.43 | 1021 | 3910 | 5454 | 1267 | 1316 | 1683 | 4183 | 1267 | 1316 | 1683 | 4183 |
| 12 | 3421 | 0.34 | 17678 | 76161 | 103427 | 22817 | 27217 | 51750 | 653233 | 34067 | 51434 | 136434 | 924550 |
| 13 | 4236 | 0.38 | 32085 | 111307 | 135504 | 32599 | 41167 | 112683 | 957066 | 49616 | 93433 | 225933 | 1371850 |
| 14 | 95 | 0.44 | 197 | 1218 | 1504 | 450 | 617 | 717 | 750 | 450 | 617 | 717 | 750 |
| 15 | 87 | 0.48 | 389 | 1513 | 1685 | 683 | 733 | 883 | 983 | 683 | 733 | 883 | 983 |
| 16 | 87 | 0.48 | 389 | 1513 | 1685 | 700 | 733 | 884 | 950 | 700 | 783 | 884 | 950 |
| total | 16570 | | 92044 | 364563 | 481778 | 106683 | 135852 | 262749 | 2815814 | 152017 | 282668 | 564234 | 387[illegible] |
| (% for total) | | | 19.1% | 75.7% | 100% | 40.6% | 51.7% | 100% | 1071.7% | 27.0% | 50.1% | 100% | 68[illegible] |
| | | | | | | 3.8% | 4.8% | 9.3% | 100% | 3.9% | 7.3% | 14.5% | 100% |

Table 6.1: Comparison of four methods - Number of Copies and CPU user time.

the Q-D and Karttunen's schemes. Since failures are found somewhere in the middle of the graphs, a 25 percent reduction in wasted copies seems reasonable with the observed unification success rate. The reduction should be smaller with higher USrate and greater with lower USrate. The substantial reduction of copies created in the QDS scheme shows the significance of structure-sharing. Somewhat uneven behaviour of the QDS scheme reflects the variety of linguistic phenemena covered in the 16 sentences. However, overall, the QDS scheme creates significantly less copies than both Q-D (Karttunen) and Wroblewski's schemes. The copies created by QDS amount to only 25 percent[5] of the copies created by the Q-D and Karttunen schemes and 19 percent of the copies created by Wroblewski's scheme.

Figure 6-2 represents the plotting of the parsing time based on the Table 6-1. Figure 6-3 is the plotting of parsing time excluding Karttunen's algorithm. The gabarge collection time is

[5] These figures are derived from the figures in the table.

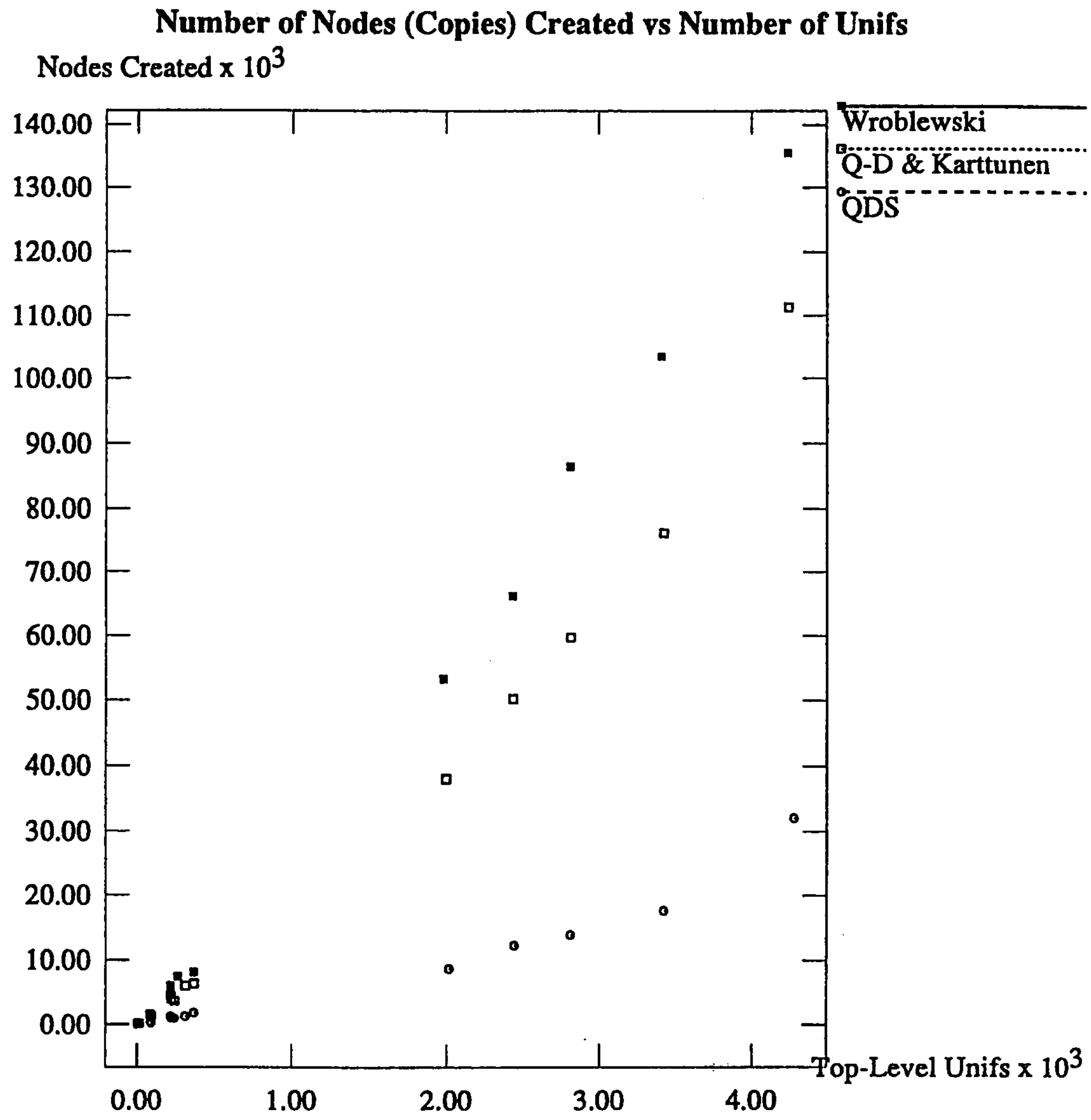


Figure 6-1: **Number of Copies vs Number of Unifications**

not included in the CPU time.

Figure 6-4 and Figure 6-5 are plottings of parsing time including the time required for garbage collection which is performed in the background. Therefore, the graphs plot the actual parsing time which is required for the parse of sample sentences. Because of the significant savings of wasted copies, the QDS runs significantly faster than the other algorithms. It only requires 3.9 percent of Karttunen's algorithm and 27.0 percent of Wroblewski's algorithm.

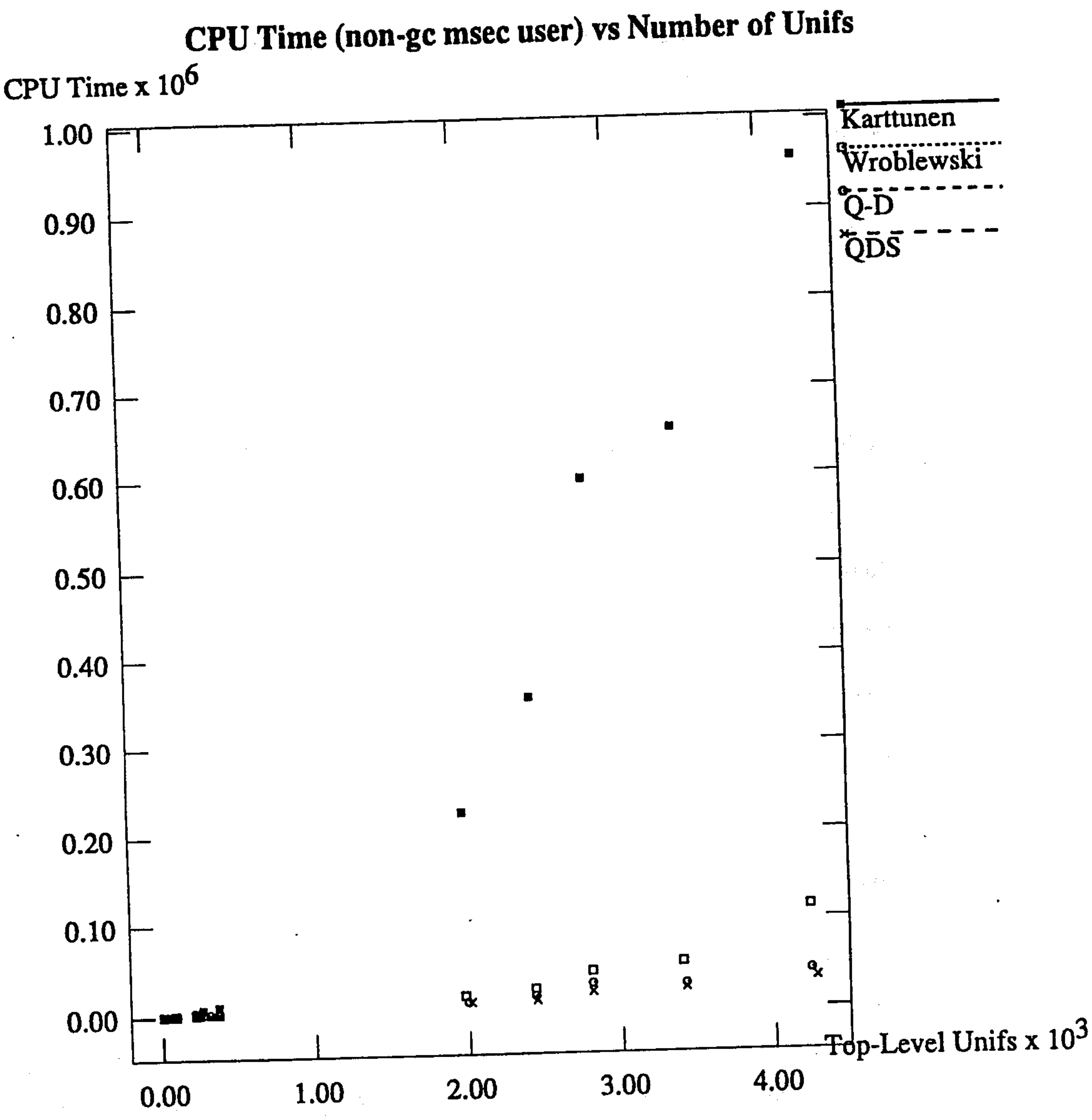


Figure 6-2: CPU time vs Number of Unifications

## 6.2 Comparison using a simulated grammar

We have seen in the previous section that the QDS algorithm runs at about 25 times the speed of Karttunen's and 4 times the speed of Wroblewski's algorithms. Also, the Q-D algorithm runs at over 10 times the speed of Karttunen's algorithm and 2 times the speed of Wroblewski's algorithm. The speed was obtained based upon the sample grammar which provides the unification success rate (USRate) of about 40 to 48 percent with the 16 sample sentences. As we discussed earlier, the strength of the Q-D algorithms depends largely on the levels of USRate. With the

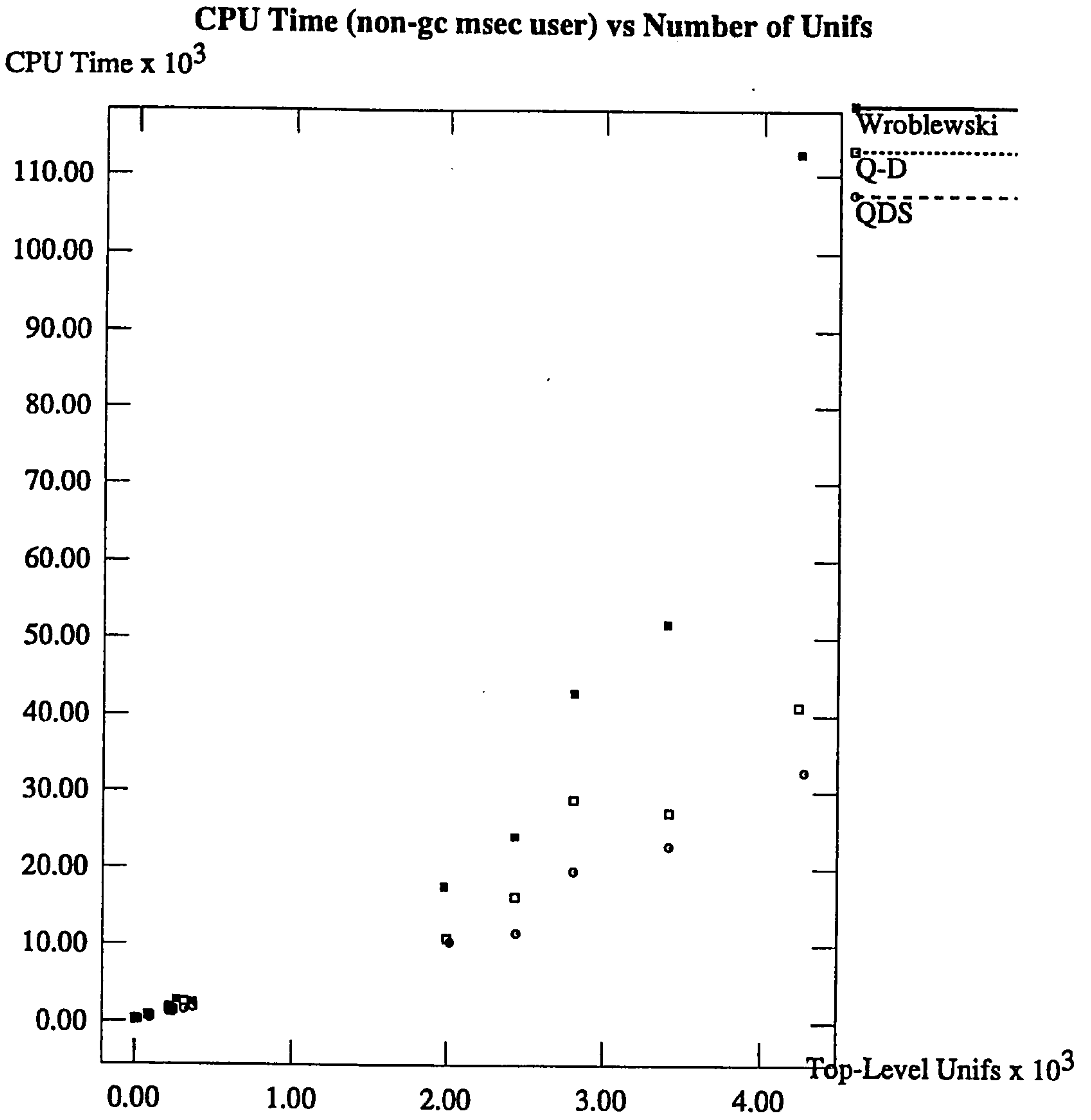


Figure 6-3: **CPU time vs Number of Unifications (without Karttunen's algorithm)**

lower USRate, the strength of the Q-D algorithms should be more conspicuous compared to the incremental algorithms since significantly fewer copies get wasted due to unification failures.

In this section, we would like to examine the behaviour of the Q-D schemes under different USRates. Since existing grammars normally produce consistent USRates for different sentences, we needed to simulate different USRates using an artificial grammar. Appendix III shows the rules and definitions we used for the experiments. In order to simulate different USRates we first define three simple rules (based upon HPSG/JPSG framework) as seen in Appendix III. Rule1 is

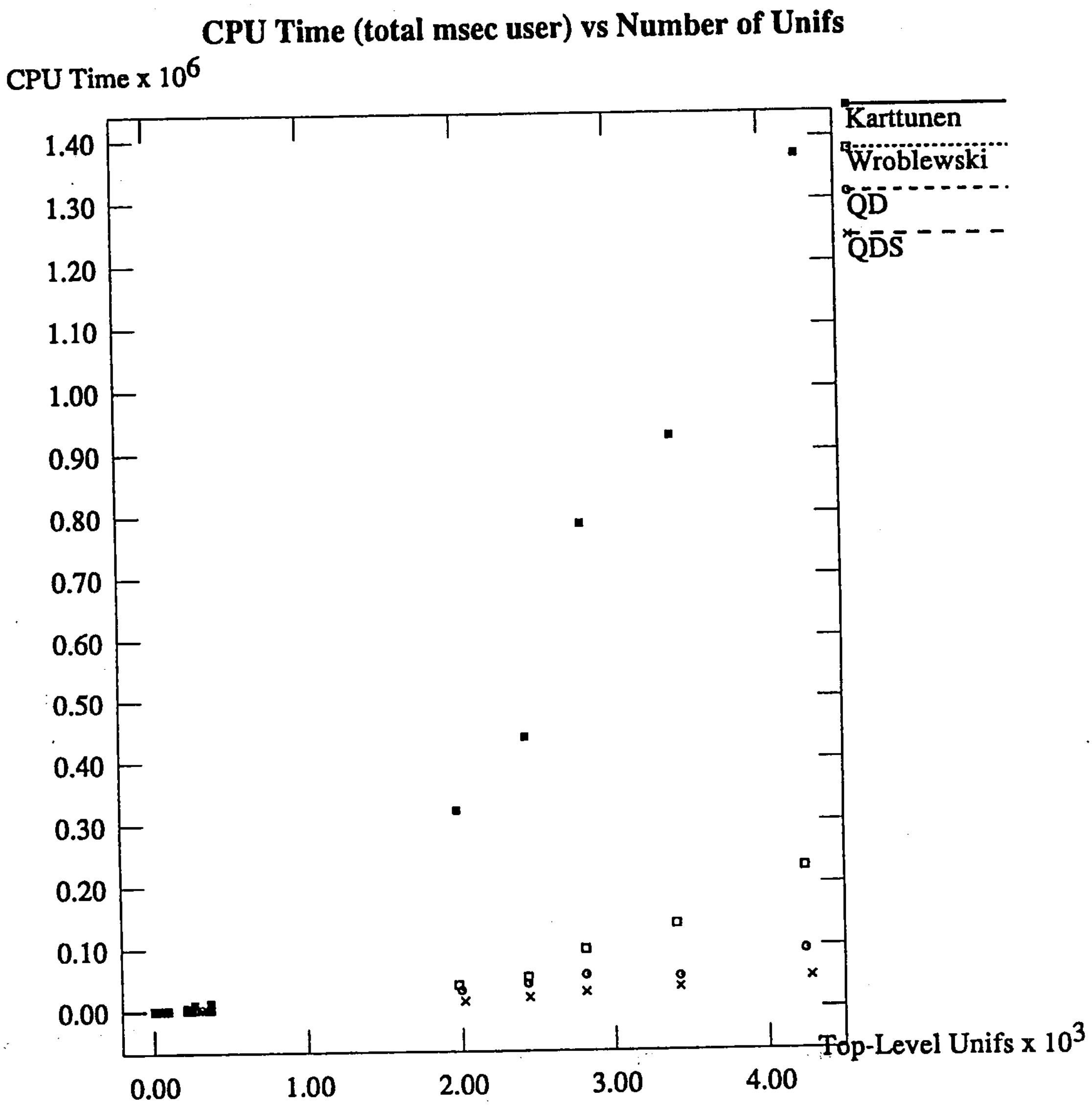


Figure 6-4: **CPU time including the background GC time.**

the Head-Feature Principle, Rule2 is the Subcat Principle and Rule3 is the Adjunct Principle. The 'rule->graph' function in the Appendix reads the PATR-II like rules and produces the directed graphs. Therefore, the rules look as below using our notation (taken from the actual output of the system):

```
rule1:
X01[[DTRS  X02[[DTR1  X03[[SYN  X04[[HEAD  X05[]]]]]
     [SYN  X06[[HEAD  X05]]
```

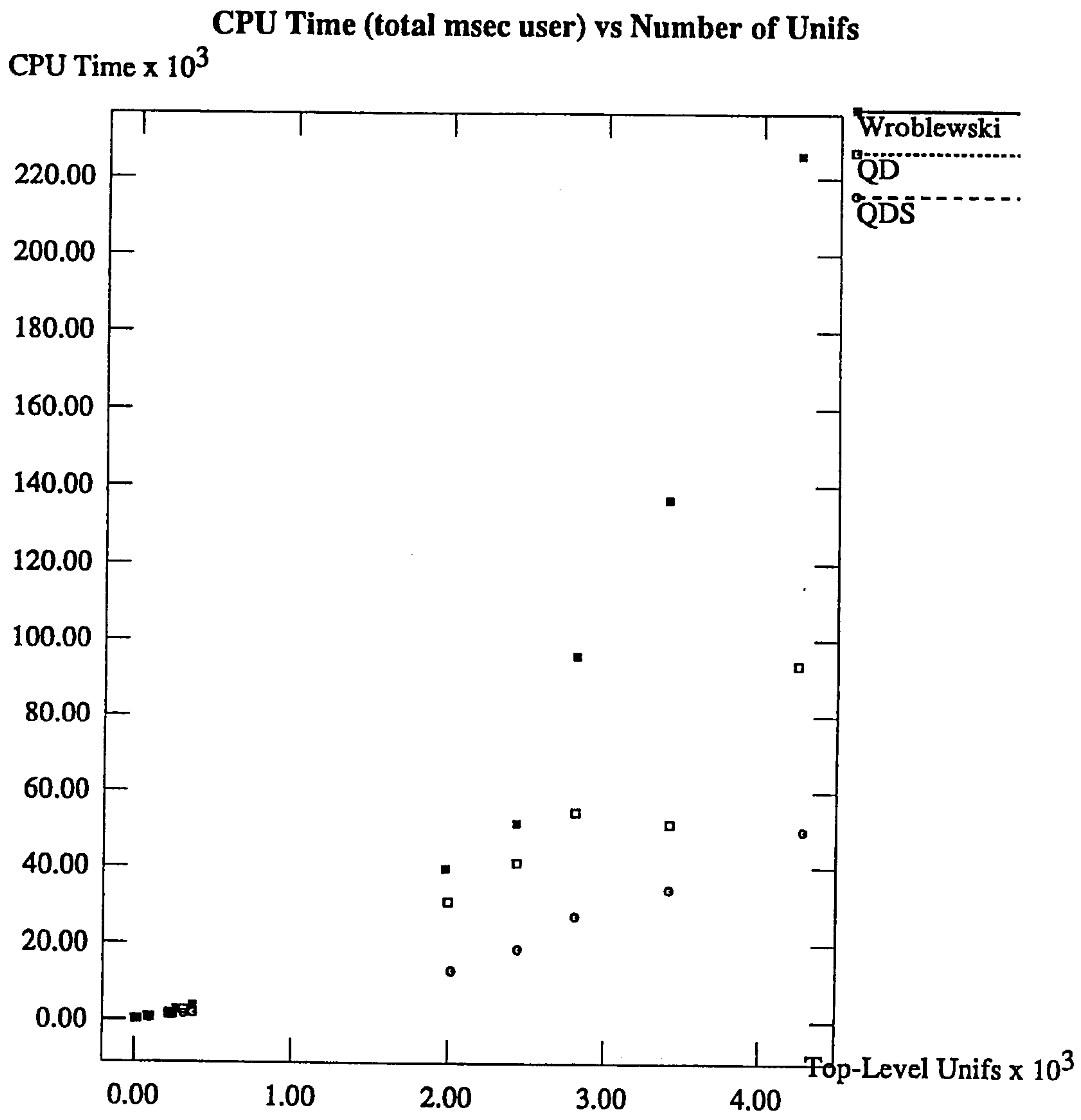


Figure 6-5: **CPU time (incl. GC) vs Number of Unifications (without Karttunen's algorithm)**

```
rule2:
X01[[DTRS  X02[[DTR1  X03[[SYN  X04[[HEAD  X05[[COH  X06[]]]]]
                [DTR2  X07[[SYN  X08[[SUBCAT  X09[[FIRST  X06]
                                                  [REST  X10[]]]]]]]
     [SYN  X11[[SUBCAT  X10]]
```

```
rule3:
X01[[DTRS  X02[[DTR1  X03[[SYN  X04[[HEAD  X05[[COH  X06[]]]]]
                [DTR2  X06]]
```

We combine these three rules, i.e., (unify-dg (unify-dg rule1 rule2) rule3)), and produce one rule graph dg1 which is:

```
dg1:
X01[[DTRS  X02[[DTR1  X03[[SYN  X04[[HEAD  X05[[COH  X06[[SYN  X07[[SUBCAT  X08[[FIRST  X06]
                                                                               [REST  X09[]]]]]]]]

               [DTR2  X06]]
    [SYN  X10[[HEAD  X05]
              [SUBCAT  X09]]
```

Note that dg1 represents the three basic principles of JPSG/HPSG and therefore characterizes typical unification-based grammar rules which are used many times during a parse. Note also that dg1 is cyclic. Given that the cycle is the result of combining the principles, the application of the cyclic rules are a common occurance using the JPSG/HPSG grammar formalisms like this one.

Now, we provide two lexical entries which are mutually exclusive. (Unification between them would fail.)

```
lex1:
X01[[SYN  X02[[HEAD  X03[[AGR  X04[[GEN  X05 FEM]
                                   [NUM  X06 SING]
                                   [PERS  X07 THIRD]]
                         [CASE  X08 -MINIATIVE]
                         [MAJ  X09 N]
                         [NFORM  X10 NORMAL]
                         [PRED  X11 MINUS]]]

lex2:
X01[[SYN  X02[[HEAD  X03[[AGR  X04[[GEN  X05 FEM]
                                   [NUM  X06 SING]
                                   [PERS  X07 THIRD]]
                         [CASE  X08 OBJECTIVE]
                         [MAJ  X09 N]
                         [NFORM  X10 NORMAL]
                         [PRED  X11 MINUS]]]
```

We unify dg1 with lex1 and get dg2:

```
dg2:
X01[[DTRS  X02[[DTR2  X03[[SYN  X04[[SUBCAT  X05[[REST  X06[]]
                                                 [FIRST  X03]]]]
               [DTR1  X07[[SYN  X08[[HEAD  X09[[COH  X03]
                                               [PRED  X10 MINUS]
                                               [NFORM  X11 NORMAL]
                                               [MAJ  X12 N]
                                               [CASE  X13 -MINIATIVE]
                                               [AGR  X14[[PERS  X15 THIRD]
                                                         [NUM  X16 SING]
                                                         [GEN  X17 FEM]]]]]]
    [SYN  X18[[SUBCAT  X06]
              [HEAD  X09]]
```

We also unify lex2 with dg1 and get dg3:

```
dg3:
X01[[DTRS  X02[[DTR2  X03[[SYN  X04[[SUBCAT  X05[[REST  X06[]]
                                                 [FIRST  X03]]]]
               [DTR1  X07[[SYN  X08[[HEAD  X09[[COH  X03]
                                               [PRED  X10 MINUS]
                                               [NFORM  X11 NORMAL]
                                               [MAJ  X12 N]
                                               [CASE  X13 OBJECTIVE]
                                               [AGR  X14[[PERS  X15 THIRD]
                                                         [NUM  X16 SING]
                                                         [GEN  X17 FEM]]]]]]
    [SYN  X18[[SUBCAT  X06]
              [HEAD  X09]]
```

The experiment (as provided in Appendix III) is as follows: We successfully unify dg1 with dg2. We unsuccessfuly unify dg2 with dg3. We plot the relation between the CPU time (non-gc) and the number of top-level unifications for different numbers of top-level unifications. (Namely, 10, 100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, and 102400 times.) We collect data for USRate 0.0, 0.25, 0.5, 0.75, and 1.0. When all unifications are between dg2 and dg3 then the USRate is 0.0. If one out of four is between dg2 and dg3 and the rest are between dg1 and dg2 then the USRate is 0.25. The described experiment was performed for 5 different kinds of USRates. We compared the Q-D algorithm and Wroblewski's algorithm. We did not compare QDS since due to structure-sharing, a fair comparison cannot be made. It is because in this experiment, same graphs are unified many times and therefore the subgraphs will simply be shared.using the QDS. Thus, the QDS would not even need to unify the subgraphs by simply returning '*T* when 'eq' (or $\equiv_r$ ) holds.

Figures 6-6 to 6-10 show the results of the experiments[6] under different USRates. One thing to be noted is that the Q-D algorithm runs faster than Wroblewski's even with the 100 percent unification success rate. It is probably because Wroblewski's algorithm needs two set-difference operations (complementarcs) in order to create copies incrementally. Also, handling cycles in Wroblewski's algorithm is adding a small amount of overhead to his algorithm.

[6] Experiments were conducted using CMU-CommonLisp run on an IBM RT with 12 mega bytes of RAM. The rules and the code provided in Appendix III are put in public domain. The code can be used as a bench mark test of the future graph unification algorithms. It is ideal for testing the behaviour under heavy use of cyclic feature structures as well as standard acyclic feature structures.

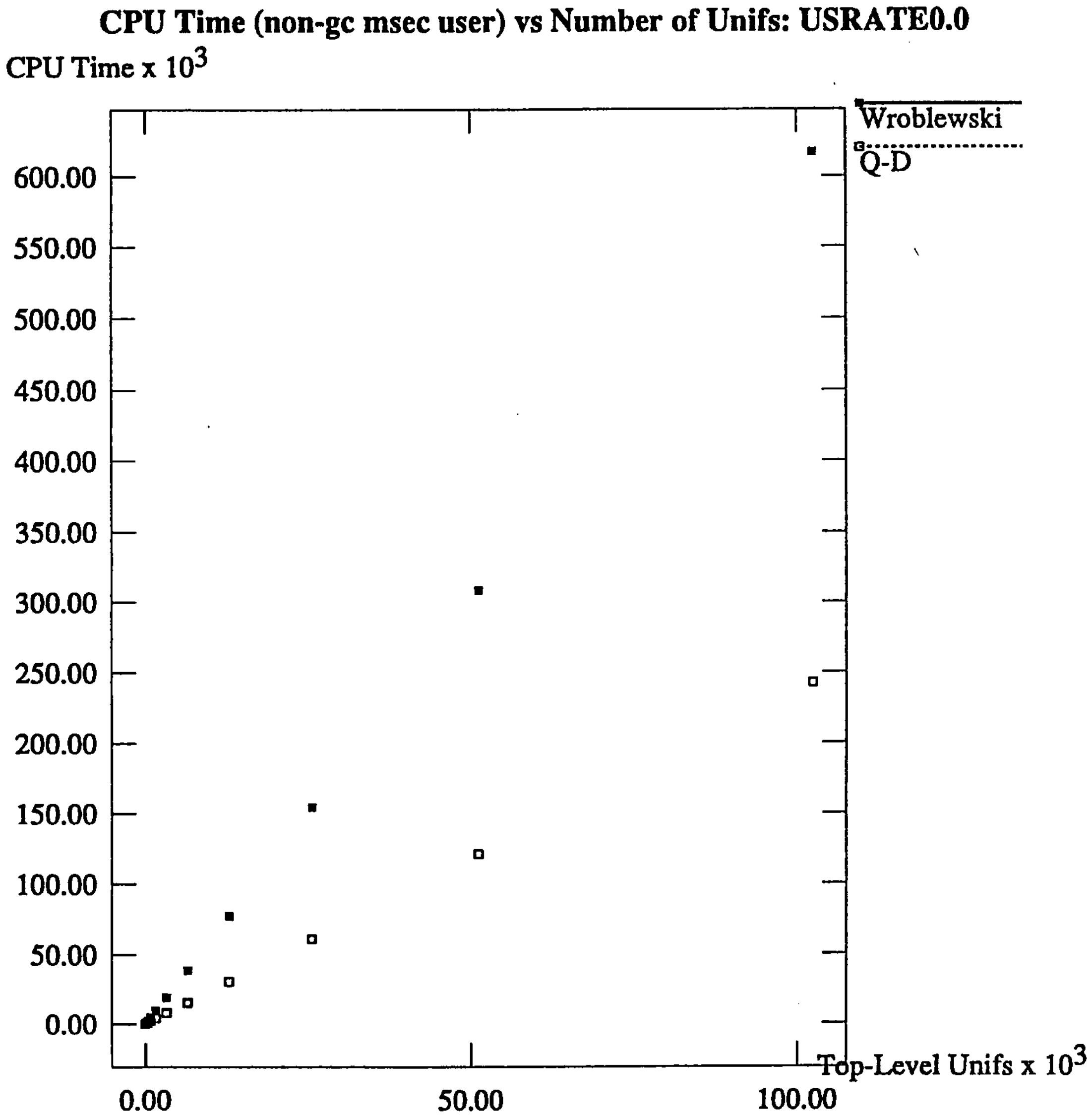


Figure 6-6: CPU time vs Number of Unifications (USRATE 0.0)

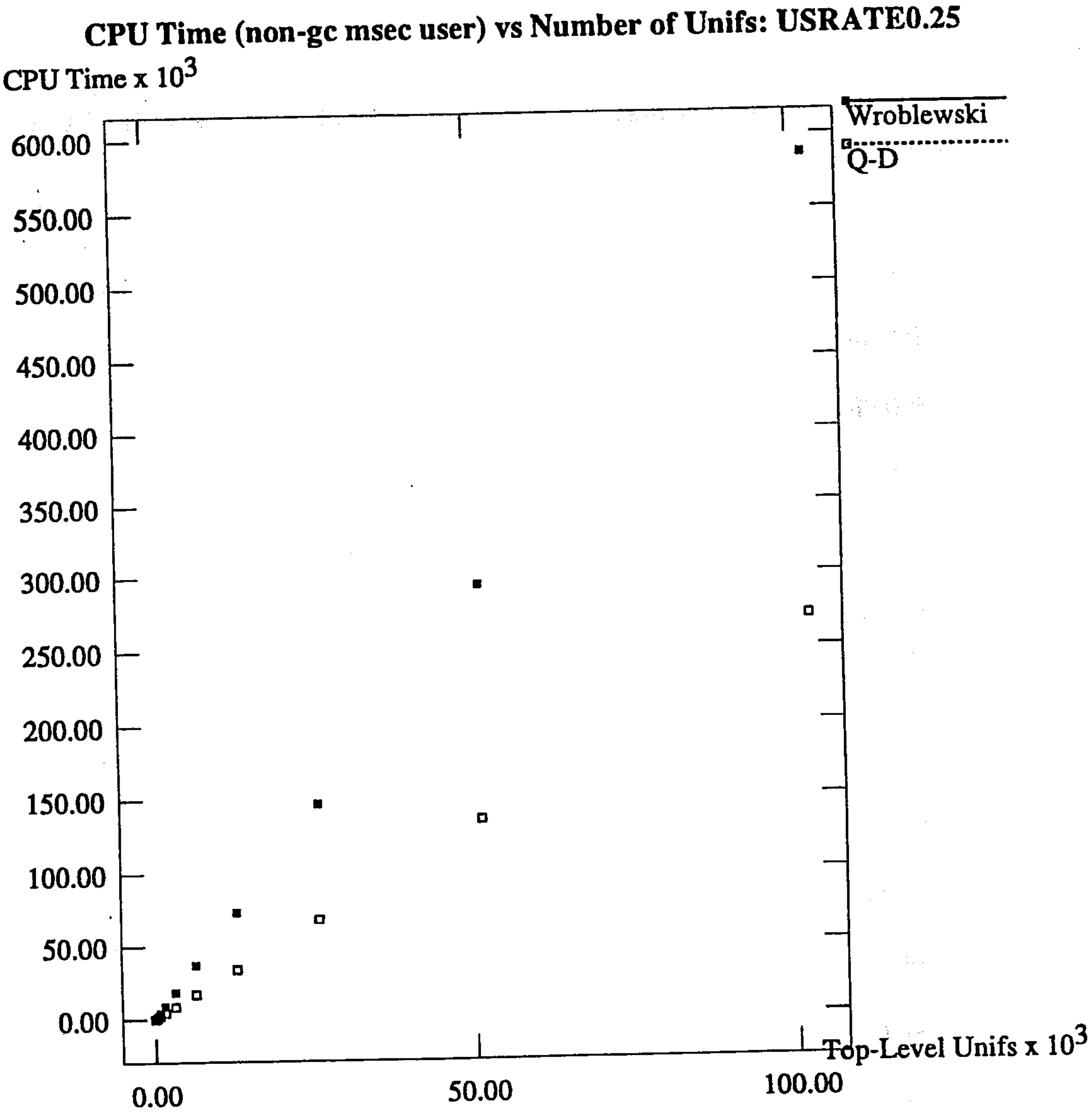


Figure 6-7: CPU time vs Number of Unifications (USRATE 0.25)

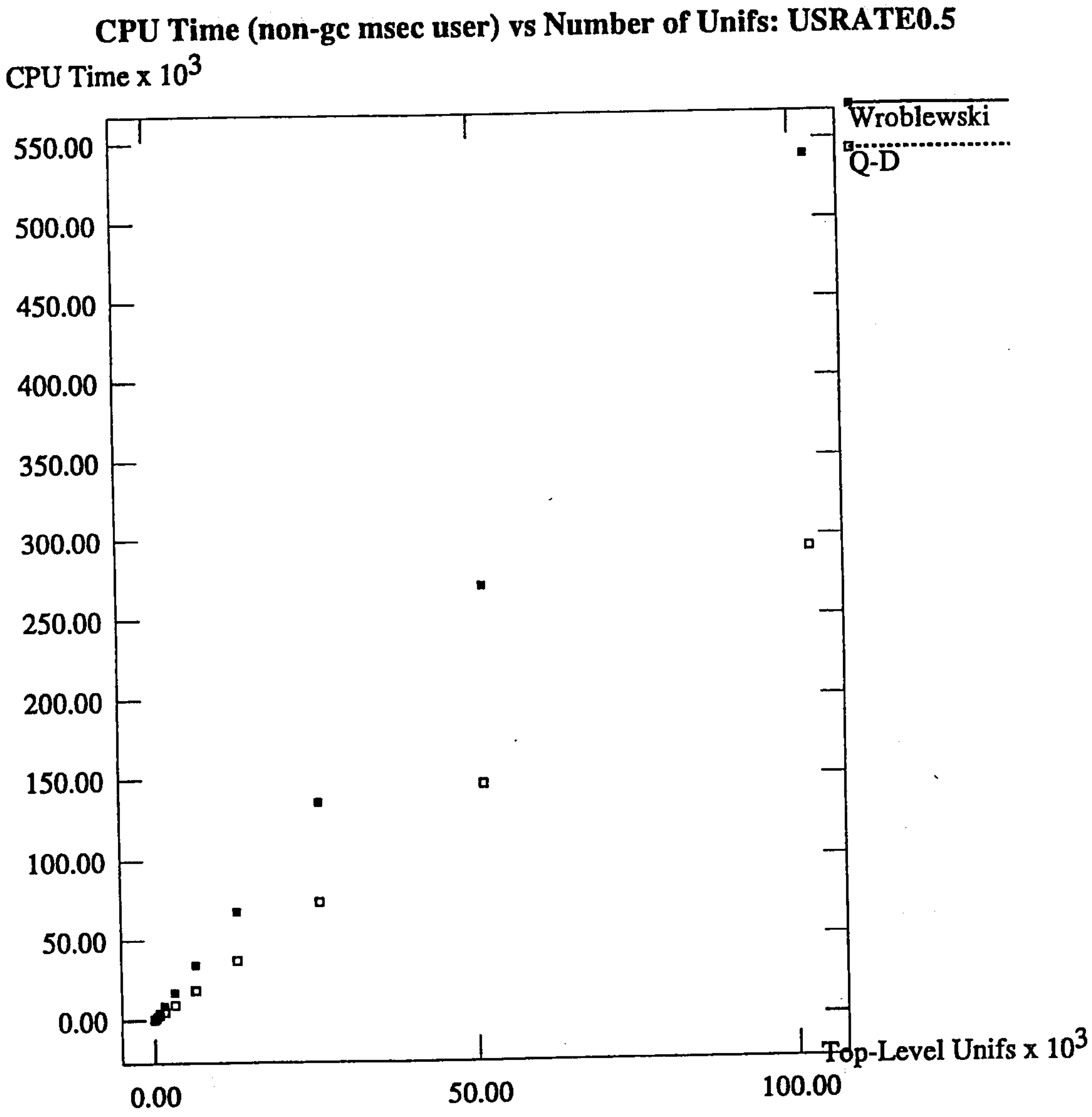


Figure 6-8: CPU time vs Number of Unifications (USRATE 0.5)

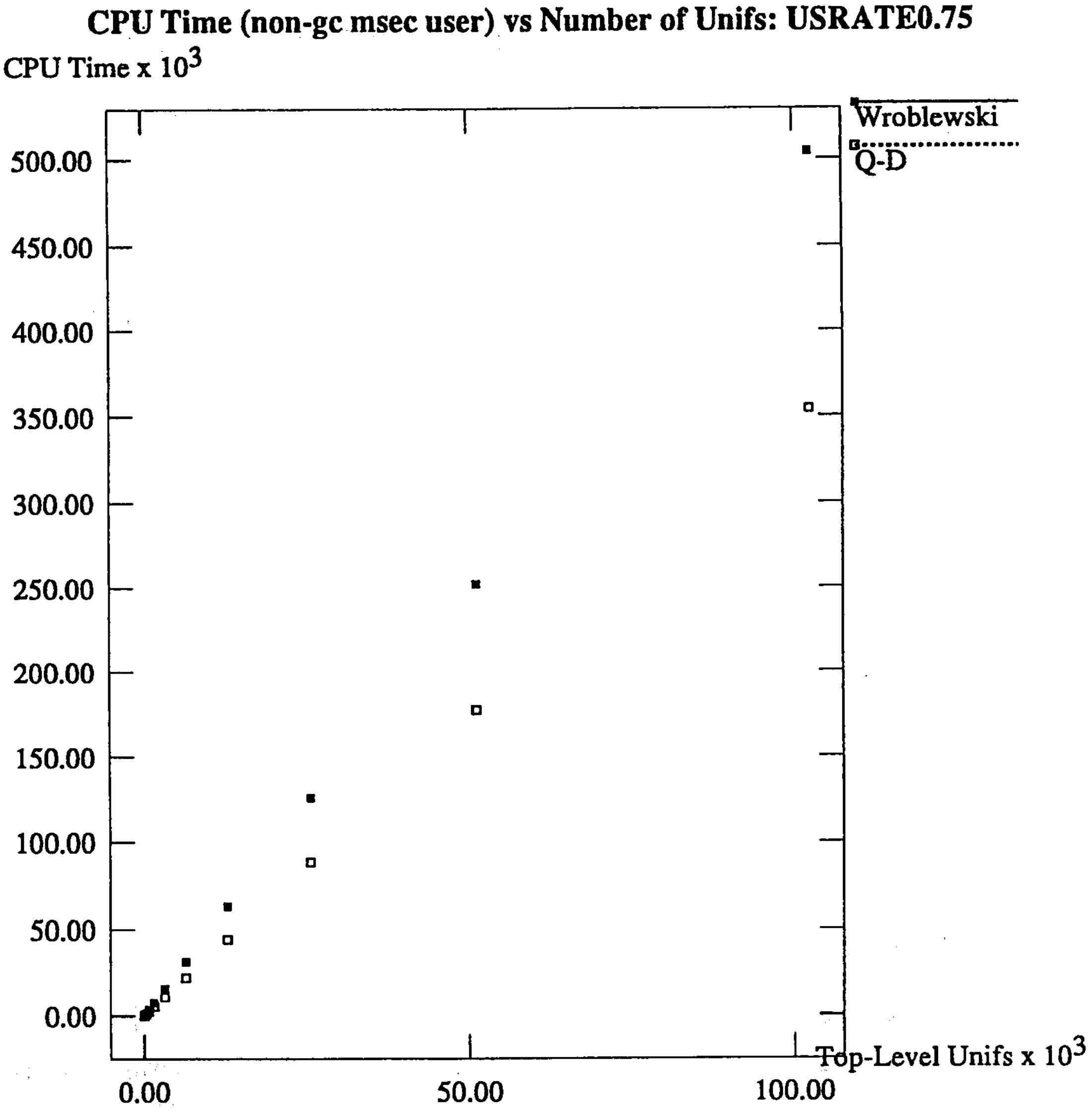


Figure 6-9: CPU time vs Number of Unifications (USRATE 0.75)

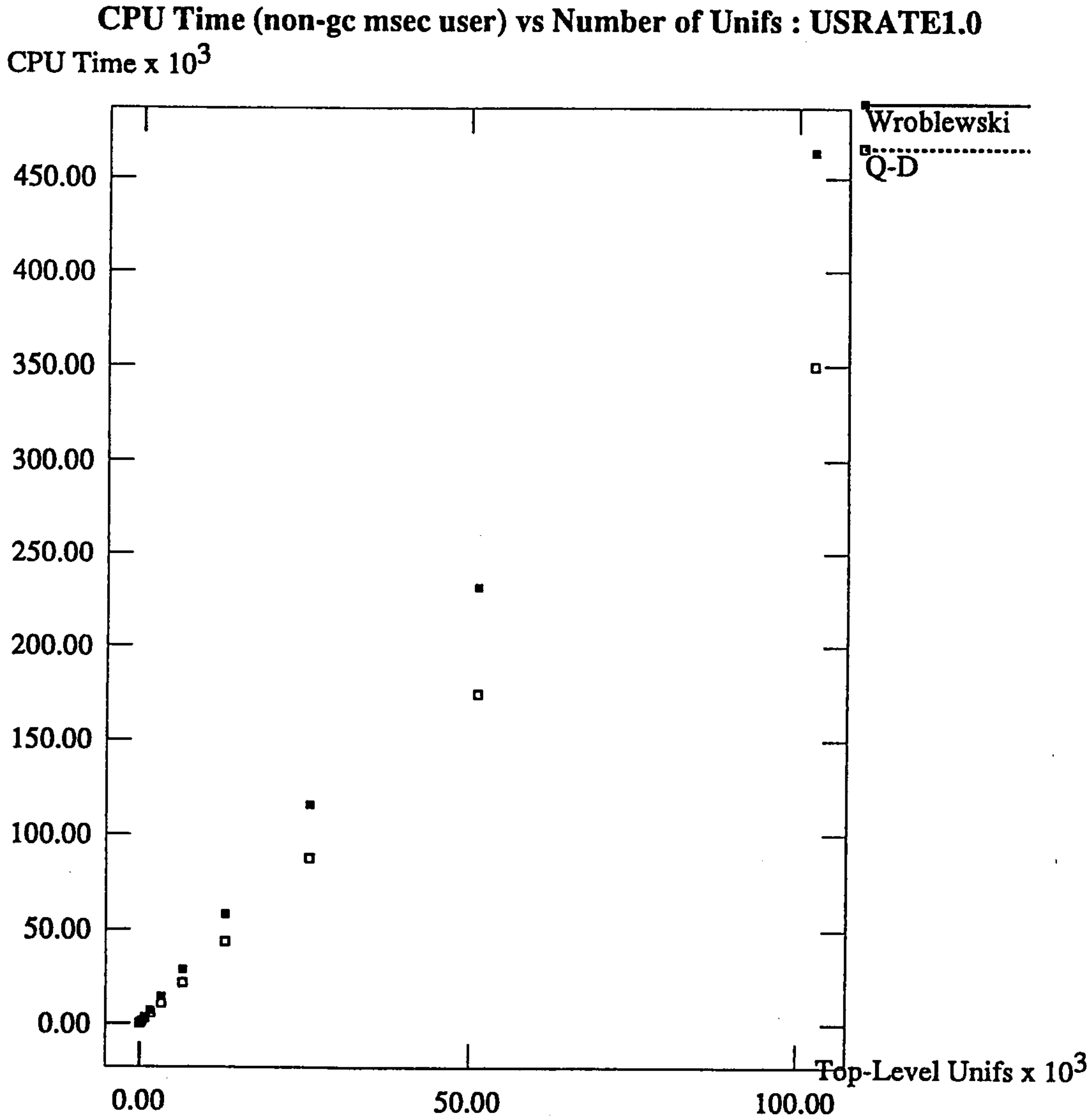


Figure 6-10: CPU time vs Number of Unifications (USRATE 1.0)

# Chapter 7

# Concluding Remarks

Unification-based constraint processing has become a *de facto* standard of natural language processing. Unification-based postulation has been accepted as the central tool for representing constraints in modern theoretical and computational linguistics. In massively parallel natural language processing, graph unification can be adopted to remedy the weakness of so-called marker-passing methods in processing syntactic constraints. In fact, Q-D algorithm was developed during the course of such a massively-parallel natural language research, in which we needed an effectively parallelizable unification ([Tomabechi, 1991b], [Tomabechi, 1991c]). Given that recursions into shared arcs were parallel-processed in the parallel-processing environment, the problem of early copying inherent in incremental schemes were devastating in the parallel unification environment. Although the topic of parallel unification is not the scope of this thesis, [Fujioka, *et al*, 1990] describes some results in parallizing the Q-D algorithm. Since the Q-D algorithm performs constraint checking without the burden of copying, we found that unification failure can be found extremely quickly by parallely spawning the recursinve unify1s deep into the feature structures. The well-known weakness of unification-based natural language processing has been slowness of speed due to the time required by unification algorithms. Given that more than 90 percent and often as much as 98 percent of parsing time is consumed by graph unification alone, the speed-up effect of improving graph unification algorithms should naturally have a greater impact than the effect of improving the speed of parsing methodologies alone. Yet, although there has been some successful and important research in speeding up parsing algorithms (such as [Tomita, 1985]), efforts to improve unification algorithms were relatively rare compared to parsing research efforts. Perhaps the reason for this could be that most natural language systems to date did not contain a very large grammar and, therefore, the performance bottleneck by unification algorithms remained largely unnoticed. Thus, it is not surprising that some of the important unification-based research came from places such as SRI, MCC, ATR and CMU, where large-scale natural language processing projects were being conducted. One early and important research effort in the feature-structure unification-based method was by Pereira. Actually, the control structure of our unify1 is similar to that of Pereira[1985]. It is the data structure of our scheme that contributes to the avoidance of the log(d) overhead that his algorithm inevitably produces to assemble feature structure by looking at the skeleton and the environment. In the proposed scheme, instead of storing changes to the argument graphs in the environment, we store the changes in the graph structures themselves (non-destructively); therefore, there will be no overhead associated with node accesses.

We share the principle of storing changes in a restorable way with Karttunen's reversible unification and we copy graphs only after a successful unification. In Karttunen's method, whenever a destructive change is about to be made, the attribute value pairs[1] stored in the body of the node are saved into an array. These values are restored after the top level unification is completed. (A copy is made prior to the restoration operation if the unification was a successful one.) Thus, in Karttunen's method, each node in the entire argument graph that

[1] I.e., arc structures: 'label' and 'value' pairs in our vocabulary.

has been destructively modified must be restored separately by retrieving the attribute values saved in an array and by resetting the values into the dag structure skeletons saved in another array. In the Q-D method, one increment to the global counter can invalidate all of the changes made to the nodes. There is also a cost for *reversing* the unification operation every time unification is completed; this cost is also proportionate to the size of the input graph. Thus, if the input graph grows (which is likely with a large-scale system) then the cost for saving and reversing changes can be high. There is also a hidden cost of Karttunen's method associated with the use of global arrays to store changes. It is the cost associated with resizing the arrays which are used to store the original information. These global arrays for saving require original allocation of memory. If the allocated memory is too big, then we will be wasting the unused memory cells; if it is too small then there will be dynamic array resizing operations during unification which can be costly. Since the number of destructive operations during unification varies significantly from sentence to sentence and from grammar to grammar, determining the desirable initial array size for Karttunen's scheme is non-trivial.

In the delayed schemes, [Karttunen and Kay, 1985] considered the use of lazy evaluation to delay destructive changes during unification. [Godden, 1990] presented one method to delay copying until a destructive change is about to take place. Godden uses delayed closures to directly implement lazy evaluation during unification. While it may be conceptually straightforward to take advantage of delayed evaluation functionalities in programming languages, actual efficiency gain from such a scheme may not be significant. This is so because such a scheme simply shifts the time and space consumed for copying to creating and evaluating closures and no significant saving can be expected overall.[2] Additionally, [Emele, 1991] also identifies

[2] Instead creating closures in comparison to creating copies could be costly in actual implementations since 'defstruct' operation which is normally used to create nodes (copies) is often effectively optimized in many commercial compilers while delayed closures are seldom optimized.

a source of other problem in Godden's method in the operations which are needed to search for already existing instances of active data structures in the copy environment and merging of environments for successive unification causing an additional overhead. Kogure and Emele also use the lazy evaluation idea to delay destructive changes. Both Kogure and Emele avoid direct usage of delayed evaluation by using pointer operations. Kogure's method also requires special dependency information to be maintained; this adds an overhead along with the cost of traversing the dependency arcs. Also, a second traversal of the set of dependent nodes is required for actually performing the copying. Emele proposes a method of dereferencing by adding environment information that carries a sequence of generation counters so that a specific generation node can be found by traversing the forwarding links until a node with that generation is found. While this allows undoing destructive changes cheaply by backtracking the environment, every time a specific graph is to be accessed the whole graph needs to be reconstructed by following the forwarding pointers sequentially, as specified in the environment list (except for the root node), in order to find the node that shares the same generation number as the root node. Therefore the overhead for dereferencing the environmental chain could be steep if a grammar is very large and if the same graphs are unified many times to create a large constituent.

Like Wroblewski's method, all three lazy methods (i.e, Godden's, Kogure's and Emele's) suffer from the problem of *Early Copying* as defined in the thesis. This is so because the copies that are incrementally created up to the point of failure during the same top-level unification are all wasted. Since the future unification result of other non-deterministic recursion into shared arcs is unknown at the point of a particular successful recursion into one shared arc, if unification succeeds with the arc, then copies are created. If a failure is detected later in some other recursive unification into the shared arcs, then the copies created until that point will all get wasted. We have explained that if we are to avoid such early copying of incremental copying schemes, then all copying must be delayed until after the entire top-level unification. That in essence is what the Q-D algorithm does. Thus, the Q-D algorithm may be seen as one extreme form of lazy copying scheme as well. The strength of it however, is that there is virtually no overhead for this full delaying of copying. The temporary forwarding pointers and comp-arc-list are utilized along with the global timing (generation) counter so that all copying can be effectively delayed until after the entire top-level unification. All changes recorded as temporary forwarding links and as updates to comp-arc-list can be invalidated very cheaply (constant time) by just one increment of the global timing counter.

The algorithm presented in this thesis has been tested using the grammar developed at CMU and at ATR and has been demonstrated to consistently run fast with large scale grammars. At CMU, the algorithm has been integrated into the new JANUS multi-language speech-to-speech translation project. Especially significant, ATR adopted the algorithm for the latest ASURA project, in which a fully separate implementation of the algorithm integrating Kogure's method for negative feature structures ([Kogure, 1992]) and Kasper's method for disjunctive feature structures ([Kasper, 1987]) was done. The preliminary data that are currently available from them (such as [Takahashi, *et al*, 1992]) has confirmed the performance of the algorithm with a very large grammar. More experiences with the algorithm should be available from other research institutes from around the world. Among them are the University of Tuebingen, the University of Karlsruhe, Keio University, Tokyo Institute of Technology, and Tokushima University who have already started using the algorithm in their projects. With the capacity to handle variables, convergence, and cycles, and with the ease of implementing it, the algorithm should be easily integrated into existing and future natural language processing mechanisms as a central constraint processing algorithm of the systems.

# Bibliography

[Bresnan and Kaplan, 1982] Bresnan, J. and R. Kaplan "Lexical-Functional Grammar: A Formal System for Grammatical Representation". In J. Bresnan (ed) *The Mental Representation of Grammatical Relations*, MIT Press, 1982.

[Earley, 1968] *An efficient context-free parsing algorithm.* Doctoral dissertation, Carnegie Mellon University.

[Emele, 1991] Emele, M. "Unification with Lazy Non-Redundant Copying". In *Proceedings of ACL-91*, 1991.

[Franz, 1990] Franz, A. *A Parser for HPSG* Report No. CMU-LCL-90-3, Carnegie Mellon University.

[Fujioka, *et al*, 1990] Fujioka, T., Tomabechi, H., Furuse, O., and H. Iida *Parallelization Technique for Quasi-Destructive Graph Unification Algorithm*, 90-NL-80, Information Processing Society of Japan, 1990.

[Gazdar, *et al*, 1985] Gazdar, G., G. Pullum, and I. Sag *Generalized Phrase Structure Grammar*. Harvard University Press, 1985.

[Gazdar and Mellish, 1989] Gazdar, G., and C. Mellish *Natural Language Processing in Lisp.* Addison Wesley, 1989.

[Godden, 1990] Godden, K. "Lazy Unification" In *Proceedings of ACL-90*, 1990.

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

[Jackendoff, 1977] Jackendoff, R. *X-Bar Syntax: A Study of Phrase Structure*, MIT Press, 1977.

[Karttunen, 1986a] Karttunen, L. *D-PATR: A Development Environment for Unification-Based Grammars*. Report CSLI-86-61. Center for the Study of Language and Information, 1986.

[Karttunen, 1986b] Karttunen, L. "D-PATR: A Development Environment for Unification-Based Grammars". In *Proceedings of COLING-86*, 1986. (Also, Report CSLI-86-61 Stanford University).

[Karttunen and Kay, 1985] Karttunen, L. and M. Kay. "Structure Sharing with Binary Trees". In *Proceedings of ACL-85*, 1985.

[Kasper, 1987] Kasper, R. "A Unification Method for Disjunctive Feature Descriptions". In *Proceedings of ACL-87*, 1987.

[Kay, 1984] Kay, M. "Functional Unification Grammar: A Formalism for Machine Translation." In *Proceedings of COLING'84*, 1984.

[Kogure, 1989] Kogure, K. *A Study on Feature Structures and Unification.* ATR Technical Report. TR-1-0032, 1988.

[Kogure, 1990] Kogure, K. "Strategic Lazy Incremental Copy Graph Unification". In *Proceedings of COLING-90*, 1990.

[Kogure, 1992] Kogure, K. "A Treatment of Negative Description of Typed Feature Structurees ". In *Proceedings of COLING-92*, 1992.

[Morimoto, *et al*, 1990] Morimoto, T., H. IIda, A. Kurematsu, K. Shikano, and T. Aizawa. "Spoken Language Translation: Toward Realizing an Automatic Telephone Interpretation System". In *Proceedings of InfoJapan 1990*, 1990.

[Pereira and Shieber, 1984] Pereira, F. and S. Shieber "The Semantics of Grammar Formalisms Seen as Computer Languages". In *Proceedings of COLING84*.

[Pereira and Warren, 1980] Pereira, F. and D. Warren "Definite clause grammars for language analysis – a survey of the formalisms and a comparison with augmented transition networks." In *Artificial Intelligence* 13, 1980.

[Pereira, 1985] Pereira, F. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms". In *Proceedings of ACL-85*, 1985.

[Pollard, 1984] Pollard, C. *Generalized Phrase Structure Grammars, Head Grammars and Natural Language*, Ph.D Dissertation, Stanford University. 1984

[Pollard and Sag, 1987] Pollard, C. and I. Sag *Information-based Syntax and Semantics.* Vol 1, CSLI, 1987.

[Shieber, *et al*, 1983] Shieber, S., H. Uszkoreit, J. Robinson, and M. Tyson *The Formalism and Implementation of PATR-II.* In Research on Interactive Acquisition and Use of Knowledge. Artificial Intelligence Center, SRI International, 1983

[Shieber, 1986] Shieber, S. *An Introduction to Unification-based Approaches to Grammar* CSLI Lecture Notes Number 4, Center for the Study of Language and Information. 1986.

[Takahashi, *et al*, 1992] Takahashi, M., H. Matsuo, K. Sagi, T. Tashiro, and M. Nagata "A Structure sharing method for Unification of Cyclic Feature Structures". In Proceedings of the National Conference of Information Processing Society of Japan, Fall, 1992.

[Tomabechi, 1991a] Tomabechi, H. "Quasi-Destructive Graph Unification". In *Proceedings of ACL-91*, 1991.

[Tomabechi, 1991b] "A Graph Propagation Architecture for Massively-Parallel Processing of Natural Language". In *Proceedings of The Thirteenth Annual Conference of the Cognitive Science Society*, 1991.

[Tomabechi, 1991c] Tomabechi, H. "MONA-LISA: Multimodal Ontological Neural Architecture for Linguistic Interactions and Scalable Adaptations". In *Proceedings of the International Workshop on Future Generation Natural Language Systems*, 1991.

[Tomabechi, 1992] Tomabechi, H. "Quasi-Destructive Graph Unification with Structure-Sharing". In *Proceedings of COLING92*, 1992.

[Tomabechi and Levin, 1989] Tomabechi, H. and L. Levin "Head-driven Massively-parallel Constraint Propagation: Head-features and subcategorization as interacting constraints in associative memory", In *Proceedings of The Eleventh Annual Conference of the Cognitive Science Society*, 1989.

[Tomita, 1985] *An efficient context-free parsing algorithm for natural languages and its applications* Doctoral dissertation, Carnegie Mellon University.

[Tomita and Carbonell, 1987] Tomita, M. and J. Carbonell. "The Universal Parser Architecture for Knowledge-Based Machine Translation". In *Proceedings of IJCAI87*, 1987.

[Tomita and Knight, 1987] Tomita, M. and K. Knight *Pseudo-Unification and Full-Unification* CMU-CMT-88-MEMO, Carnegie Mellon University, 1987.

[Warren, 1983] Warren, D. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, 1983,

[Wroblewski, 1987] Wroblewski, D."Nondestructive Graph Unification". In *Proceedings of AAAI87*, 1987.

[Yoshimoto and Kogure, 1989] Yoshimoto, K. and K. Kogure *Japanese Sentence Analysis by means of Phrase Structure Grammar.* ATR Technical Report. TR-1-0049, 1989.

# Appendix I: Sample Sentences

SENT1 "MOSHIMOSHI"
'Hello.'

SENT2 "SOCHIRAHATSUUYAKUDENWAKOKUSAIKAIGIJIMUKYOKUDESUKA"
'Is [polite] this [hearer] the secretariat (for the)
International Interpreting Telephony Conference?'

SENT3 "HAI"
'Yes.'

SENT4 "SOUDESU"
'it is [polite]'.

SENT5 "WATASHIHAKAIGINIMOUSHIKOMITAINODESUGA"
'I would like to register for (the) conference.'

SENT6 "TOUROKUYOUSHIHAARIMASUKA"
'Do (you) have [polite] (the) registration form.'

SENT7 "IIE"
'No.'

SENT8 "WAKARIMASHITA"
'(I) understand [polite].'

SENT9 "SOREDEHAKOCHIRAKARASOCHIRANITOUROKUYOUSHIWOOOKURIITASHIMASU"
'Then, we [speaker] will send [polite] you the [hearer]
(the)registration form.'

SENT10 "ONAMAETOGOZYUUSYOWOONEGAISHIMASU"
'May (I) ask (your) name [polite] and address [polite], please?'

SENT11 "OOSAKASHIKITAKUCYAYAMACHI6NO23*SUZUKIMAYUMIDESU"
'Oosaka city, Kitakucyaya town, 6-23, Suzuki Mayumi, it is [polite].'

SENT12 "KOCHIRAKARASOCHIRANITOUROKUYOUSHIWOSHIKYUUNIOOKURIITASHIMASU"
'We [speaker] will send [polite] you [hearer] (the)
registration form immediately.'

SENT13 "WAKARANAITENGAGOZAIMASHITARAWATAKUSHIDOMONIITSUDEMOOKIKIKUDASAI"

'If there is [polite] (a) question, ask [respect] us [polite] anytime.'

SENT14 "ARIGATOUGOZAIMASU"
'Thank (you) very much.'

SENT15 "SOREDEHASHITSUREISHIMASU"
'Then, good bye [polite].'

SENT16 "DOUMOSHITSUREISHIMASU"
'Thank you and good bye [polite].'

# Appendix II: Sample Grammar

This is a representative portion of the grammar used in the experiments reported in Chapter 6. The grammar is based upon ATR grammar ([Yoshimoto and Kogure, 1989]) using JPSG/HPSG analysis.

```
(rule p ==> (n p)
  (<0 head>    == <2 head>)
  (<0 subcat>  == <2 subcat rest>)
  (<0 slash>   == <1 slash>)
  (<1>         == <2 subcat first>)
  (<0 wh>      == <1 wh>)
  (<0 sem>     == <2 sem>)
  (<0 semf>    == <2 semf>)
  (<0 prag>    == <1 prag>))

(rule v ==> (p v)
  (<0 head>          == <2 head>)
  (<0 subcat>        == <2 subcat rest>)
  (<0 slash>         == <2 slash>)
  (<1>               == <2 subcat first>)
  (<2 wh wh-ind>     == -)
  (<0 wh>            == <1 wh>)
  (<0 sem>           == <2 sem>)
  (<1 head form>     == (:or ga wo ni kara))
  (<0 prag speaker>        == <1 prag speaker>)
  (<0 prag speaker>        == <2 prag speaker>)
  (<0 prag hearer>         == <1 prag hearer>)
  (<0 prag hearer>         == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))

(rule v ==> (p v)
  (<0 head>                == <2 head>)
  (<0 subcat>              == <2 subcat rest>)
  (<0 slash>               == <2 slash>)
  (<1>                     == <2 subcat first>)
  (<2 wh wh-ind>           == +)
  (<0 wh>                  == <2 wh>)
  (<0 sem>                 == <2 sem>)
  (<1 head form>           == (:or ga wo ni to))
  (<0 prag speaker>        == <1 prag speaker>)
  (<0 prag speaker>        == <2 prag speaker>)
  (<0 prag hearer>         == <1 prag hearer>)
  (<0 prag hearer>         == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))

(rule n ==> (p n)
  (<1 head coh>   == <2>)
  (<0 head>       == <2 head>)
  (<1 head form>  == (:or to *))
  (<0 subcat>     == <2 subcat>)
  (<0 foot slash> == <2 foot slash>)
  (<0 sem reln>   == <1 sem reln>)
  (<0 sem arg-1>  == <1 sem arg-1>)
  (<0 sem arg-2>  == <2 sem>))

(rule n ==> (v n)
  (<0 head>             == <2 head>)
  (<1 head cform>       == adnm)
  (<1 subcat>           == end)
  (<1 slash first sem>  == <2 sem>)
  (<1 slash first semf> == <2 semf>)
  (<0 slash>            == <1 slash rest>)
  (<2 sem>              == ?noun_sem)
  (<1 sem>              == ?verb_sem)
  (<0 sem>              == [[parm ?noun_sem]
    [restr ?verb_sem]])
  (<0 prag>             == <1 prag>))

(rule v ==> (v)
  (<0 head ctype>    == <1 head ctype>)
  (<1 lex>           == +)
  (<0 head>          == <1 head>)
  (<0 subcat>        == <1 subcat rest>)
  (<0 slash rest>    == <1 slash>)
  (<0 slash first>   == <1 subcat first>)
  (<0 wh>            == <1 wh>)
  (<0 sem>           == <1 sem>)
  (<1 subcat first>  == [[head [[pos p]]]])
  (<0 prag>          == <1 prag>))

(rule v ==> (v auxv)
  (<0 head>             == <2 head>)
  (<0 subcat>           == <2 subcat rest>)
  (<0 slash>            == <1 slash>)
  (<1>                  == <2 subcat first>)
  (<0 sem>              == <2 sem>)
  (<0 prag speaker>     == <1 prag speaker>)
  (<0 prag speaker>     == <2 prag speaker>)
  (<0 prag hearer>      == <1 prag hearer>)
  (<0 prag hearer>      == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))

(rule v ==> (v vinfl)
  (<0 head>             == <2 head>)
  (<0 subcat>           == <1 subcat>)
  (<0 slash>            == <1 slash>)
  (<1 head cform>       == stem)
  (<2 lex>              == -)
  (<1>                  == <2 subcat first>)
  (<2 subcat rest>      == end)
  (<0 wh>                  == <1 wh>)
  (<0 sem>                 == <2 sem>)
  (<0 prag speaker>        == <1 prag speaker>)
  (<0 prag speaker>        == <2 prag speaker>)
  (<0 prag hearer>         == <1 prag hearer>)
  (<0 prag hearer>         == <2 prag hearer>)
  (<0 prag restr first>    == <1 prag restr>)
  (<0 prag restr rest>     == <2 prag restr>))

(rule v ==> (p v)
  (<2>              == <1 head coh>)
  (<0 head>         == <2 head>)
  (<1 head form>    == (:or ha mo))
  (<0 subcat>       == <2 subcat>)
  (<0 slash>        == <2 slash rest>)
  (<0 sem>          == <2 sem>)
  (<1 sem>          == <2 slash first sem>)
  (<1 semf>         == <2 slash first semf>)
  (<0 wh>           == <2 wh>)
  (<0 prag speaker>        == <1 prag speaker>)
  (<0 prag speaker>        == <2 prag speaker>)
  (<0 prag hearer>         == <1 prag hearer>)
  (<0 prag hearer>         == <2 prag hearer>)
  (<0 prag restr first>    == <2 prag restr>)
  (<0 prag restr rest>     == <1 prag restr>))

(rule n ==> (v n)
  (<0 head>         == <2 head>)
  (<0 subcat>       == <2 subcat rest>)
  (<1>              == <2 subcat first>)
  (<1 head cform>   == adnm)
  (<2 head form>    == no)
  (<0 slash>        == <1 slash>)
  (<0 wh>           == <1 wh>)
  (<0 sem>          == <1 sem>)
  (<0 prag>         == <1 prag>))

(rule v ==> (n v)
  (<0 head>         == <2 head>)
  (<0 subcat>       == <2 subcat rest>)
  (<1>              == <2 subcat first>)
  (<2 head modl>    == [[copl +]])
  (<0 slash>        == <1 slash>)
  (<0 wh>           == <1 wh>)
  (<0 sem>          == <2 sem>)
  (<0 prag speaker>        == <1 prag speaker>)
  (<0 prag speaker>        == <2 prag speaker>)
  (<0 prag hearer>         == <1 prag hearer>)
  (<0 prag hearer>         == <2 prag hearer>)
  (<0 prag restr first>    == <2 prag restr>)
  (<0 prag restr rest>     == <1 prag restr>))

(rule v ==> (adv v)
  (<0 head>                == <2 head>)
  (<0 subcat>              == <2 subcat rest>)
  (<1>                  == <2 subcat first>)
  (<2 head modl>        == [[copl +]])
  (<0 slash>            == <1 slash>)
  (<0 wh>               == <1 wh>)
  (<0 sem>              == <2 sem>)
  (<0 prag speaker>     == <1 prag speaker>)
  (<0 prag speaker>     == <2 prag speaker>)
  (<0 prag hearer>      == <1 prag hearer>)
  (<0 prag hearer>      == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))

(rule v ==> (adv v)
  (<2>                  == <1 head coh>)
  (<0 head>             == <2 head>)
  (<0 subcat>           == <2 subcat>)
  (<0 slash>            == <2 slash>)
  (<2 wh wh-ind>        == -)
  (<0 wh>               == <1 wh>)
  (<0 sem>              == <2 sem>)
  (<0 prag speaker>     == <1 prag speaker>)
  (<0 prag speaker>     == <2 prag speaker>)
  (<0 prag hearer>      == <1 prag hearer>)
  (<0 prag hearer>      == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))

(rule v ==> (v auxv)
  (<0 head>        == <2 head>)
  (<0 subcat>      == <2 subcat rest>)
  (<1>             == <2 subcat first>)
  (<0 slash>       == <1 slash>)
  (<1 wh wh-ind>   == +)
  (<0 wh>          == end)
  (<2 head modl>   == [[sfp-1 ka]])
  (<0 sem>         == <2 sem>)
  (<0 prag speaker>     == <1 prag speaker>)
  (<0 prag speaker>     == <2 prag speaker>)
  (<0 prag hearer>      == <1 prag hearer>)
  (<0 prag hearer>      == <2 prag hearer>)
  (<0 prag restr first> == <2 prag restr>)
  (<0 prag restr rest>  == <1 prag restr>))
```

# Appendix III: A Bench-Mark Code to Produce Simulated Grammar

The rule definitions below provide simple definitions of Head-Feature Principle, Subcat Principle and Adjunct Principle. When Subcat Principle and Adjunct Principle are combined a cycle will result. The following code provides a simulated grammatical analysis by combining these rules. The code will simulate different unification success rates (0.0, 0.25, 0.5, 0.75 and 1.0). This code is used to produce the graphs provided in Figure 6-6 to Figure 6-10 in Chapter 6. These rule definitions and the code are put in the public domain. This code should be useful to be used as a benchmark test of unification algorithms.

```
;;; -*- Mode: Lisp; Syntax: Common-lisp; Package: User; Base: 10 -*-    ;;;
;;; Copyright (C) 1993 by Hideto Tomabechi

;;; head feature principle
(setq rule1 (rule->graph
 '(((syn head) = (dtrs dtr1 syn head)))
  ))

;;; subcat principle
(setq rule2 (rule->graph
  '(((syn subcat) = (dtrs dtr2 syn subcat rest))
    ((dtrs dtr1 syn head coh) = (dtrs dtr2 syn subcat first)))
  ))

;;; adjunct principle
(setq rule3 (rule->graph
  '(((dtrs dtr1 syn head coh) = (dtrs dtr2)))
  ))

;;; combined rule.
(setq dg1 (unify-dg (unify-dg rule1 rule2) rule3))

(setq lex1 (rule->graph
  '(((syn head maj) = N)
    ((syn head nform) = normal)
    ((syn head agr pers) = third)
    ((syn head agr num) = sing)
    ((syn head agr gen) = fem)
    ((syn head pred) = minus)
    ((syn head case) = -miniative))))
```

```
(setq lex2 (rule->graph
  '(((syn head maj) = N)
    ((syn head nform) = normal)
    ((syn head agr pers) = third)
    ((syn head agr num) = sing)
    ((syn head agr gen) = fem)
    ((syn head pred) = minus)
    ((syn head case) = objective))))

(setq dg2 (unify-dg lex1 dg1))

(setq dg3 (unify-dg lex2 dg1))

;;;; simple data gathering below:
(defparameter *times* 10)
(proclaim '(fixnum *10*))

(defun data10 (&optional (times *times*))
  "Usrate 1.0"
  (declare (type fixnum times)
   (special *times* *dgnodes* *dgarcs*))
  (format t "~% ~%Unification for ~a times." times)
  (format t "~% USRate 1.0 ~%")
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (time (dotimes (n times)
        (unify-dg dg1 dg2)))
  (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
  (format t "~% Number of Arcs  Created:  ~A" *dgarcs*))

(defun data0 (&optional (times *times*))
  "Usrate 0.0"
  (declare (type fixnum times)
   (special *times* *dgnodes* *dgarcs*))
  (format t "~% Unification for ~a times." times)
  (format t "~% USRate 0.0 ~%")
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (time (dotimes (n times)
     (unify-dg dg2 dg3)))
  (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
  (format t "~% Number of Arcs  Created:  ~A" *dgarcs*))


(defun data5 (&optional (times *times*))
  "Usrate 0.5"
  (declare (type fixnum times)
   (special *times* *dgnodes* *dgarcs*))
  (format t "~% Unification for ~a times." times)
  (format t "~% USRate 0.5 ~%")
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (time (dotimes (n (truncate (/ times 2)))
     (declare (type fixnum n))
    (unify-dg dg1 dg2)
    (unify-dg dg2 dg3)))
  (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
  (format t "~% Number of Arcs  Created:  ~A" *dgarcs*))

(defun data25 (&optional (times *times*))
  "Usrate 0.25"
  (declare (type fixnum times)
   (special *times* *dgnodes* *dgarcs*))
  (format t "~% Unification for ~a times." times)
  (format t "~% USRate 0.25 ~%")
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (time (dotimes (n (truncate (/ times 4)))
    (declare (type fixnum n))
    (unify-dg dg2 dg3)
    (unify-dg dg3 dg2)
    (unify-dg dg2 dg3)
    (unify-dg dg1 dg2)))
  (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
  (format t "~% Number of Arcs  Created:  ~A" *dgarcs*))


(defun data75 (&optional (times *times*))
  "Usrate 0.75"
  (declare (type fixnum times)
   (special *times* *dgnodes* *dgarcs*))
  (format t "~% Unification for ~a times." times)
  (format t "~% USRate 0.75 ~%")
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (time (dotimes (n (truncate (/ times 4)))
    (declare (type fixnum n))
    (unify-dg dg1 dg2)
    (unify-dg dg1 dg3)
    (unify-dg dg2 dg1)
    (unify-dg dg2 dg3)))
  (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
  (format t "~% Number of Arcs  Created:  ~A" *dgarcs*))
```

# Appendix IV: Sample Code

What follows is the sample implementation of the Q-D algorithms using CommonLisp. The code has been tested on Allegro, Lucid, and CMU CommonLisp. The code has been maintained for two years since the initial implementation conducted for International Workshop on Parsing Technologies 1990, and then for ACL91 and COLING92. Currently the code is running with stability. This code is available via email or in magnetic forms. Contact me at: tomabech@cs.cmu.edu, tomabech@is.tokushima-u.ac.jp, or tomabech@mtlab.sfc.keo.ac.jp.

```
;;; -*- Mode: Lisp; Syntax: Common-lisp; Package: User; Base: 10 -*-    ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                                                                     ;;;
;;;                QUASI-DESTRUCTIVE UNIFICATION ALGORITHM              ;;;
;;;                        (Q-D and QDS versions)                       ;;;
;;;                                                                     ;;;
;;; Copyright (C) 1990, 1993 by Hideto Tomabechi.  All rights reserved. ;;;
;;;                                                                     ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                       ;;
;;                                                                       ;;
;;-----------------------------------------------------------------------;;
;;                                                                       ;;
;; File Created:   11-July-90 by tomabech                                ;;
;; Last Edit Date: 17-Jan -93 by tomabech                                ;;
;;                                                                       ;;
;;-----------------------------------------------------------------------;;
;;;=====================================================================
;;;
;;;     Global Variables
;;;
(defparameter *quasi-version* 5.3 "version of the algorithm implementation")

(setq *features* (adjoin :TOMABECHI *features*))

(defparameter *atom-sharing* nil "if non-nil, perform structure-sharing for
                                atoms in the non-structure sharing mode")

(defparameter *str-sharing* t "If non-nil, use structure sharing scheme.
                               Structure sharing is performed in the
                               following way.  Atomic and Leaf nodes can
                               be shared. Complex nodes can be shared if
                               no nodes below was changed (forwarding or
                               comp-arc-list). If a node is forwarded or
```

```
                                  has comp-arc-list that information is passed
                                  up using multiple-value-bind facility when
                                  copy of nodes and arcs are passed up. When
                                  a node is atomic or leaf and was not being
                                  forwarded to them, they are regarded unchanged
                                  and the original nodes are shared. When they
                                  are destination of forwardeding then they are
                                  considered changed. The original nodes are
                                  shared by putting it in a copy arc and the
                                  nodes above must be copied. (change information
                                  passed up). If a node is a complex and is not
                                  a destination of forwarding, then if a
                                  comp-arc-list exists, then it is copied and
                                  changed information is passed up the recursions;
                                  if no comp-arc-list does not exist, then the
                                  original node is returned and unchanged info is
                                  passed up.  If the complex node is a target of
                                  forwarding, the same happens but 'change' info
                                  is always passed up.  This method enables the
                                  structure sharing of nodes that are unchanged.")

(defparameter *unification* 'quasi-unify "name of unification algorithm")

(proclaim '(type t *atom-sharing* *str-sharing*))

(defvar *debug-stream1* *standard-output*)
(defvar *dgnode-list* nil)
(defvar *unify-global-counter* 10) ;;; start from 10
(proclaim '(fixnum *unify-global-counter*))

(defvar *dgnodes* 0 "to count number of dgnodes created for experiments.")
(defvar *dgarcs* 0  "to count number of dgarcs created for experiments.")
(defvar *unify0* 0   "to count number of unify0s called.")
(defvar *unify1* 0   "to count number of unify1s called.")
(proclaim '(fixnum *dgnodes* *dgarcs* *unify0* *unify1*))

;;;=====================================================================


;;;
;;; Inline declaration
;;;
#-symbolics
(proclaim '(inline make-dgnode create-dgnode complementarcs intersectarcs
    copy-node-comp-not-forwarded copy-node-comp-forwarded
            complement-arcs intersect-arcs add-arc-to-dgnode change-dgnode-type
    get-dgnode create-complex-dgnode create-atomic-dgnode  ;;; below is for
    create-leaf-dgnode set-dgnode-to-arc-list add-feature  ;;; Kogure.
    get-dgnode-from-arc-list get-value-from-arc get-feature-from-arc
    dgnode-type-of dgnode-typep copy-unify1-dgnode copy-unify1-node
    print-dgnode print-arc find-real-result-dgnode
    pprint-fs-internal2 pprint-fs-leaf pprint-fs-atomic
    pprint-fs-complex
            equal-dg
    change-to-atomic-dgnode change-to-complex-dgnode
    atomic-dgnode-p complex-dgnode-p leaf-dgnode-p
```

```
   ))

;;;==================================================================
;;; Data Structure
;;;
;;;     Definition of DGNODE
;;;
#-lucid
(defstruct (DGNODE (:print-function pprint-dg))  ;;; this for long mesg
;;;(defstruct (DGNODE (:print-function kprint-dgnode)) ;;; this for short mesg
   (type nil :type symbol)
   (arc-list nil :type list) ;;; content of arc-list is an atomic value if type is :atomic
   (comp-arc-list nil :type list)
   (forward nil :type dgnode)
   (copy nil :type dgnode)
   (generation 0 :type fixnum)
   mark ;;; for Kevin's grammar reader (unused).
   )

#+lucid
(defstruct (DGNODE (:print-function pprint-dg))  ;;; this for long mesg
;;;(defstruct (DGNODE (:print-function kprint-dgnode)) ;;; this for short mesg
   (type nil :type symbol)
   (arc-list nil :type (or atom list)) ;;; content of arc-list is an atomic value if type is :atomic
   (comp-arc-list nil :type list)
   (forward nil :type (or atom dgnode))
   (copy nil :type (or atom dgnode))
   (generation 0 :type fixnum)
;;;   mark ;;; for Kevin's grammar reader (unused).
   )

;; ARC-TYPES
;;; currently unused for experiments
(defconstant *normal* '=)
(defconstant *must-be-present* '=c)
(defconstant *multiple-valued* '>)

;;
;;; ==================================================================
;;;; DGNODE CREATION

(eval-when (compile load eval)
  (deftype dgarc () 'cons)
 )

(defmacro create-arc (&key (label nil)
                           (type *normal*)
           (value nil))
 (declare (type symbol label)
          (type symbol type)
  (type dgnode value)
  (special *dgarcs*))
 `(progn
   (incf *dgarcs*)
   (cons ,label ,value)))
```

```
(defmacro arc-label (arc) '(car ,arc))
(defun arc-label-func (arc) (arc-label arc))
(defmacro arc-value (arc) '(cdr ,arc))
(defmacro arc-p (arc) '(consp ,arc))

(defun create-dgnode (&key (type :atomic)
    (arc-list nil)
                            ;;; (mark nil) ;;; CMU
    )
  (declare (type symbol type)
   (type list arc-list))
  (let ((temp (make-dgnode
         :arc-list arc-list
         :type type
;;; :mark mark ;;; CMU
)))
     (declare (type dgnode temp))
;     (incf *dgnodes*) ;;; count number of dgnodes created
     temp))

(defmacro %create-dgnode (&optional (type :atomic))
 "macro version of create-dgnode"
 (declare (type symbol type)
  (special *dgnodes*))
 '(let ((temp (make-dgnode
  :type ,type)))
;      (push temp *dgnode-list*)
    (declare (type dgnode temp))
    (incf *dgnodes*) ;;; count number of dgnodes created
     temp))

;;; ====================================================================
;;;; MACRO DEFINITIONS

(defmacro ATOMICNODE-p (dgnode)
 (declare (type dgnode dgnode))
 '(eq (DGNODE-type ,dgnode) :atomic))

(defmacro LEAFNODE-p (dgnode)
 (declare (type dgnode dgnode))
  '(eq (DGNODE-type ,dgnode) :leaf))

(defmacro COMPLEXNODE-p (dgnode)
 (declare (type dgnode dgnode))
  '(eq (DGNODE-type ,dgnode) :complex))

(defmacro find-atomic (arc-lst)
 "receives an arclist and returns the first occurance of an
  arc structure with the :atomic type."
  '(find :atomic ,arc-lst
 :test #'eq
 :key #'(lambda (arc) (arc-label arc))))

(defmacro find-leaf (arc-lst)
  '(find :leaf ,arc-lst
 :test #'eq
 :key #'(lambda (arc) (arc-label arc))))

(defmacro find-complex (arc-lst)
  '(find :complex ,arc-lst
 :test #'eq
 :key #'(lambda (arc) (arc-label arc))))

;;;; generic version
;;;(defmacro find-arc-with-label (label arc-lst)
;;; "returns the first occurrance of the arc structure with
;;;  the label in arc-lst"
;;;  '(find ,label ,arc-lst
;;;  :test #'eq
;;;  :key #'(lambda (arc) (arc-label arc))))

;;; for cons arc structure only
(defmacro find-arc-with-label (label arc-lst)
 "This one for cons only."
 (declare (type symbol label)
  (type list arc-list))
 '(assoc ,label ,arc-lst :test #'eq))

(defmacro simple-copy-arc (arc)
  (declare (type dgarc arc))
  '(create-arc :label (arc-label ,arc)
       :value (simple-copy-dgnode (arc-value ,arc))))

(defmacro identical-atomic-dgnodep (dg1 dg2)
  (declare (type dgnode dg1 dg2))
  '(eq (DGNODE-arc-list ,dg1) (DGNODE-arc-list ,dg2)))

(defmacro dgnode-arc-labels (dgnode)
  (declare (type dgnode dgnode))
  '(when (COMPLEXNODE-p ,dgnode)
          (mapcar #'(lambda (arc)
                       (arc-label arc))
                 (dgnode-arc-list ,dgnode))))

(defmacro return-real-arc (label dgnode)
 "return an arc in the dg node that is with the arc-label."
  (declare (type symbol label)
          (type dgnode dgnode))
 '(if (and (DGNODE-comp-arc-list ,dgnode)
   (= *unify-global-counter*
      (DGNODE-generation ,dgnode)))
      (or (find ,label (DGNODE-arc-list ,dgnode)
    :test #'eq
    :key #'(lambda (a) (ARC-label a)))
  (find ,label (DGNODE-comp-arc-list ,dgnode)
    :test #'eq
    :key #'(lambda (a) (ARC-label a))))
      (find ,label (DGNODE-arc-list ,dgnode)
    :test #'eq
```

```
      :key #'(lambda (a) (ARC-label a)))))


(defmacro set-temporary-forward-dgnode (dgnode1 dgnode2)
  "This is a temporary forwarding of dgnode1 to dgnode2, i.e.,
   just like in the case of copying, the value of the
   generation field must meet the *unify-global-counter*"
  (declare (type dgnode dgnode1 dgnode2)
   (special *unify-global-counter*))
  '(unless (or (eq ,dgnode1 ,dgnode2)
        (= (dgnode-generation ,dgnode1) 9)) ;;; added 10/1/91
     (setf (dgnode-forward ,dgnode1) ,dgnode2)
     (setf (dgnode-generation ,dgnode1) *unify-global-counter*)))

(defmacro set-permanent-forward-dgnode (dgnode1 dgnode2)
  "This is a permanent forwarding of dgnode1 to dgnode2, i.e.,
   standard Wroblewski type forwarding.  The mark 9 indicates
   that it is a permanent forwarding"
  (declare (type dgnode dgnode1 dgnode2))
  '(unless (eq ,dgnode1 ,dgnode2)
     (setf (dgnode-forward ,dgnode1) ,dgnode2)
     (setf (dgnode-generation ,dgnode1) 9)))

(defmacro forward-dg (dg1 dg2 &optional (type :temporary))
 "We have two kinds of forwarding: temporary and permanent. If it
  is temporary, it is only good during the same unify0. If it is
  permanent it is hardwired just as in Wroblewski's algorithm."
 (declare (type dgnode dg1 dg2)
  (type symbol type))
 '(case ,type
     (:temporary (set-temporary-forward-dgnode ,dg1 ,dg2))
     (:permanent (set-permanent-forward-dgnode ,dg1 ,dg2))))

;;; 11/16/91 macro-version
(defmacro dereference-dg (dg-input)
  "This is an iterative version of dereferenced-dg."
  (declare (type dgnode dg-input))
  '(do ((result ,dg-input dg)
 (dg ,dg-input
  (if (and (DGNODE-forward dg)
   (or (= (DGNODE-generation dg) *unify-global-counter*)
       (= (DGNODE-generation dg) 9))) ;;; 9 means permanent
      (DGNODE-forward dg)
      (setf (DGNODE-forward dg) nil)))) ;;; make it GCable and return nil
        ((null dg) result)
     (declare (type dgnode result dg))))


;;;
;;;
;;;   MAP-DOLIST is like DOLIST, except it returns a list of all results.
;;;   This macro is from Tommy's (Masaru Tomita) CMT/CMU utilities.
(defmacro map-dolist (varlist body)
 (let ((map-result (gensym)))
 '(let ((,map-result nil))
    (dolist ,varlist (push ,body ,map-result))
```

```
     (nreverse ,map-result))))

(defun complementarcs (dg1 dg2)
  "arcs that exist in dg1 but not in dg2.
   Content of comp-arc-list is respected if generation
   is valid (matches the global counter."
  (declare (type dgnode dg1 dg2)
   (special *unify-global-counter*))
  (let ((arc-list1
   (if (and (DGNODE-comp-arc-list dg1)
    (= *unify-global-counter*
       (DGNODE-generation dg1)))
       (append (DGNODE-comp-arc-list dg1)
       (DGNODE-arc-list dg1))
       (DGNODE-arc-list dg1)))
(arc-list2
   (if (and (DGNODE-comp-arc-list dg2)
    (= *unify-global-counter*
       (DGNODE-generation dg2)))
       (append (DGNODE-comp-arc-list dg2)
       (DGNODE-arc-list dg2))
       (DGNODE-arc-list dg2))))
    (declare (type list arc-list1 arc-list2))
    (set-difference arc-list1
            arc-list2
   :test
   #'(lambda (arc1 arc2)
      (eq (arc-label arc1)
  (arc-label arc2))))))

(defun intersectarcs (dg1 dg2)
  "Tail recursive. This function returns only one value,
   namely, shared arcs from dg1 only."
  (declare (type dgnode dg1 dg2))
  (labels ((tr-intersectarcs (arcs1 arcs2 result1)
     (cond ((null arcs1) result1)
   ((member (ARC-label (car arcs1))
    arcs2
    :test #'eq
    :key #'(lambda (bbb)
     (ARC-label bbb)))
    (tr-intersectarcs (cdr arcs1)
       arcs2
       (cons (car arcs1) result1)))
   (t (tr-intersectarcs (cdr arcs1)
 arcs2
 result1)))))
    (declare (type list arcs1 arcs2 result1))
    (let ((arc-list1
   (if (and (DGNODE-comp-arc-list dg1)
    (= *unify-global-counter*
       (DGNODE-generation dg1)))
       (append (DGNODE-comp-arc-list dg1)
       (DGNODE-arc-list dg1))
       (DGNODE-arc-list dg1)))
 (arc-list2
  (if (and (DGNODE-comp-arc-list dg2)
   (= *unify-global-counter*
      (DGNODE-generation dg2)))
      (append (DGNODE-comp-arc-list dg2)
      (DGNODE-arc-list dg2))
      (DGNODE-arc-list dg2))))
     (declare (type list arc-list1 arc-list2))
     (tr-intersectarcs arc-list1 arc-list2 nil))))


;;; ====================================================================
;;; ====================================================================
;;;; UNIFICATION FUNCTIONS
;;;
;;;     Unify Toplevel Function
;;;

;;; graph-unify, and unify-fs are for different parsers.

(defmacro graph-unify (dg1 dg2 &optional result)
 "This is the top-level unification function"
 '(unify-dg ,dg1 ,dg2 ,result))

(defmacro unify-fs (dg1 dg2 &optional result)
 "This is the top-level unification function"
 '(unify-dg ,dg1 ,dg2 ,result))

(defun unify-dg (dg1 dg2 &optional result)
  "This is the top-level unification function"
 (declare (type dgnode dg1 dg2))
 (setq result (catch 'UNIFY-FAIL (unify0 dg1 dg2)))
 (incf *unify-global-counter*)
 result)

 (defun unify0 (dg1 dg2)
  "If unify1 succeeds, make a copy of dg1.  Content of comp-arc-list of
  dg1 will be added to the content of arc-list of the copy."
  (declare (type dgnode dg1 dg2))
  (incf *unify0*)
  (if (eq '*T* (unify1 dg1 dg2))
      (copy-dg-with-comp-arcs dg1)))


 ;;; last mod 4/30/92 based on Marie Boyle's (nnsbo01@mailserv.zdv.uni-tuebingen.de)
 ;;; suggestion to avoid infinite loop in a successful cycle.
 (defun unify1 (dg1-underef dg2-underef)
  "intersectarcs only returns shared1 value"
  (declare (type dgnode dg1-underef dg2-underef))
  (incf *unify1*)
  (let ((dg1 (dereference-dg dg1-underef))
        (dg2 (dereference-dg dg2-underef)))
   (declare (type dgnode dg1 dg2))
   (if (DGNODE-copy dg1) ;;; this copy is not current, so get rid of it.
       (setf (DGNODE-copy dg1) nil))
  (if (DGNODE-copy dg2)
      (setf (DGNODE-copy dg2) nil))
  (cond ((eq dg1 dg2)  ;;; because of forwarding and loop, it is
        '*T*)           ;;; possible that dg1 and dg2 are eq.
        ((LEAFNODE-p dg1)
         (forward-dg dg1 dg2 :temporary) ;;; forward dg1 to dg2.
         '*T*)
        ((LEAFNODE-p dg2)
         (forward-dg dg2 dg1 :temporary) ;;; forward dg2 to dg1.
         '*T*)
        ((and (ATOMICNODE-p dg1) (ATOMICNODE-p dg2))
         (cond ((identical-atomic-dgnodep dg1 dg2)
                (forward-dg dg2 dg1 :temporary)
                '*T*)
               (t (throw 'UNIFY-FAIL nil))))
        ((or (ATOMICNODE-p dg1) (ATOMICNODE-p dg2))
         (throw 'UNIFY-FAIL nil))
        (T (let ((shared1 (intersectarcs dg1 dg2)))
             (declare (type list shared1))
             (cond ((null shared1)  ;;; no shared arc => success
                    (forward-dg dg2 dg1 :temporary)
                    (setf (DGNODE-comp-arc-list dg1) (complementarcs dg2 dg1)
                          (DGNODE-generation dg1)  *unify-global-counter*)
                    '*T*)
                   (t (forward-dg dg2 dg1 :temporary)
                      (dolist (arc1 shared1)
                        (declare (type dgarc arc1))
                        (unify1 (ARC-value arc1)
                                (ARC-value (return-real-arc
                                       (ARC-label arc1)
                                                   dg2))))
                        ;;; if all recursions succeeded then below:
                      (let ((new (complementarcs dg2 dg1)))
                        (declare (type list new))
                        (if (DGNODE-comp-arc-list dg1)
    (if (= *unify-global-counter*
                                   (DGNODE-generation dg1))
                                (dolist (newarc new)
                                  (declare (type dgarc newarc))
                                  (push newarc (DGNODE-comp-arc-list dg1)))
                                (setf (DGNODE-comp-arc-list dg1) nil))
                            (setf (DGNODE-comp-arc-list dg1) new
                                  (DGNODE-generation dg1)
                                  *unify-global-counter*)))
                      '*T*)))))))

(defun copy-dg-with-comp-arcs (dgnode-underef)
  (declare (type dgnode dgnode-underef)
   (special *str-sharing*))
  (if *str-sharing*
      (copy-dg-with-comp-arcs-share dgnode-underef)
      (copy-dg-with-comp-arcs-no-share dgnode-underef)))

(defmacro copy-arc-and-comp-arc-no-share (arc)
  (declare (type dgarc arc))
  `(create-arc  :label (arc-label ,arc)
       :value (copy-dg-with-comp-arcs-no-share (arc-value ,arc))))

(defmacro copy-arc-and-comp-arc-share (arc)
 "if destination nodes are changed than make a copy otherwise return
  the arc itself."
 (declare (type dgarc arc))
 `(multiple-value-bind (destination changed)
      (copy-dg-with-comp-arcs-share (arc-value ,arc))
    (declare (type dgnode destination)
             (type symbol changed))
    (if changed
 (values (create-arc :label (arc-label ,arc)
    :value destination)
 :changed)
 (values ,arc nil))))

(defmacro copy-arc-and-comp-arc (arc)
  (declare (special *str-sharing*)
   (type dgarc arc))
  `(if *str-sharing*
      (copy-arc-and-comp-arc-share ,arc)
      (copy-arc-and-comp-arc-no-share ,arc)))

(defmacro simple-copy-dgnode (dgnode-underef)
  (declare (special *str-sharing*)
   (type dgnode dgnode-underef))
  `(if *str-sharing*
      (copy-dg-with-comp-arcs-share ,dgnode-underef)
      (simple-copy-dg-no-share ,dgnode-underef)))


;;; original one.
(defun copy-dg-with-comp-arcs-no-share (dgnode-underef)
  "Recursively go down the dgnode and make a copy of the dg.
   Content of the comp-arc-list in the original dg should be
   put in the arc-list of the copy. Ignore the comp-arc-list with
   old generation stamp."
  (declare (type dgnode dgnode-underef)
   (special *atom-sharing*))
  (let ((dgnode (dereference-dg dgnode-underef)))
   (declare (type dgnode dgnode))
   (cond ((and (DGNODE-copy dgnode)                  ;;; if a current copy exists
      (= (DGNODE-generation (DGNODE-copy dgnode)) *unify-global-counter*))
  (DGNODE-copy dgnode))                      ;;; then return the copy
        ((ATOMICNODE-p dgnode)
  (if *atom-sharing*
      dgnode
    (let ((newdgnode (%create-dgnode)))
      (declare (type dgnode newdgnode))
      (setf
       (DGNODE-type newdgnode) :atomic
       ;; (DGNODE-name newdgnode) (DGNODE-name dgnode)
       (DGNODE-arc-list newdgnode) (DGNODE-arc-list dgnode) ;;; value
       (DGNODE-generation newdgnode) *unify-global-counter*
       (DGNODE-copy dgnode) newdgnode
      )
     newdgnode)))
((LEAFNODE-p dgnode)              ;;; Bottom of lattice, ie, Variable.
 (let ((newdgnode (%create-dgnode)))
   (declare (type dgnode newdgnode))
   (setf
     (DGNODE-type newdgnode) :leaf
;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
             (DGNODE-generation newdgnode) *unify-global-counter*
     (DGNODE-copy dgnode) newdgnode
     )
   newdgnode))
(T   ;;; complex-dgnode
 (let ((newdgnode (%create-dgnode)))
   (declare (type dgnode newdgnode))
   (setf
     (DGNODE-type newdgnode) :complex
;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
             (DGNODE-generation newdgnode) *unify-global-counter*
             (DGNODE-copy dgnode) newdgnode
    )  ;;; ^^this setf for copy is moved up from after recursion. Due to Peter, 9/3/91
   (dolist (arc (DGNODE-arc-list dgnode)) ;;;do parallel
                 (declare (type dgarc arc))
         (push (copy-arc-and-comp-arc-no-share arc)
       (DGNODE-arc-list newdgnode)))
   (if (and (DGNODE-comp-arc-list dgnode)
    (= *unify-global-counter* (DGNODE-generation dgnode)))
       (dolist (comp-arc (DGNODE-comp-arc-list dgnode)) ;;;do parallel
                     (declare (type dgarc comp-arc))
     (push (copy-arc-and-comp-arc-no-share comp-arc)
       (DGNODE-arc-list newdgnode))))
   (setf (DGNODE-comp-arc-list dgnode) nil) ;;; make it GCable 10/1/91
   newdgnode)))))

;;; mod 9/4/91 and 9/18/91 based upon Peter's bug report.
;;; mod 4/30/92 based on Takahashi.
(defun copy-node-comp-not-forwarded (dgnode)
 "When the node to be copied is not a result of forwarding then,
   we will not need pass :changed markers upward."
 (declare (type dgnode dgnode))
  (cond ((ATOMICNODE-p dgnode)
 (values dgnode nil)) ;;;the second value nil indicates no change
((LEAFNODE-p dgnode)
 (values dgnode nil))
(T   ;;; complex-dgnode
 (cond ((and (DGNODE-comp-arc-list dgnode)
     (= *unify-global-counter*
(DGNODE-generation dgnode)))
(let ((newdgnode (%create-dgnode)))
  (declare (type dgnode newdgnode))
  (setf
    (DGNODE-type newdgnode) :complex
                    (DGNODE-generation newdgnode) *unify-global-counter*
    ;; (DGNODE-name newdgnode) (DGNODE-name dgnode)
    (DGNODE-copy dgnode) newdgnode
```

```
)
(dolist (comp-arc (DGNODE-comp-arc-list dgnode))
                 (declare (type dgarc comp-arc))
  (push (copy-arc-and-comp-arc-share comp-arc)
(DGNODE-arc-list newdgnode)))
(dolist (arc (DGNODE-arc-list dgnode))
                 (declare (type dgarc arc))
  (push (copy-arc-and-comp-arc-share arc)
(DGNODE-arc-list newdgnode)))
(setf (DGNODE-comp-arc-list dgnode) nil) ;;; 10/1/91 for GC.
(values newdgnode :changed)))
    (t (let ((state nil))
                 (declare (type symbol state))
                 (setf (DGNODE-copy dgnode) dgnode
                       (DGNODE-generation dgnode) *unify-global-counter*)
  ;;; --- This hack is needed to avoid infinite loop
  ;;;  with a cyclic graph.  By making a copy to be
  ;;;  itself infinite loop can be avoided. 9/18/91 tomabech
                 (let ((arcs (map-dolist (arc (DGNODE-arc-list dgnode))
                           (multiple-value-bind (arc changed)
                              (copy-arc-and-comp-arc-share arc)
                             (declare (type dgarc arc)
                                      (type symbol changed))
                             (if changed
                                 (setq state changed))
                               arc))))
                       (cond (state
                        (cond ((not (eq (DGNODE-copy dgnode) dgnode))
                              (setf
                               (DGNODE-arc-list (DGNODE-copy dgnode)) arcs
                               (DGNODE-type (DGNODE-copy dgnode)) :complex
                               )
                              (values (DGNODE-copy dgnode) :changed))
                             (t
                              (let ((newdgnode (%create-dgnode)))
                                (declare (type dgnode newdgnode))
                                (setf
                                 (DGNODE-type newdgnode) :complex
                                 (DGNODE-generation newdgnode) *unify-global-counter*
                            ;;;;     (DGNODE-name newdgnode) (DGNODE-name dgnode)
                                 (DGNODE-arc-list newdgnode) arcs
                                 (DGNODE-copy dgnode) newdgnode
                                 )
                                (values newdgnode :changed)))))
                          (t (setf (DGNODE-copy dgnode) nil)
                          ;;; This corresponds to the above hack.
                          ;;; Reset the copy field when actually no
                          ;;; copy was made. 9/18/91 tomabech
                          (values dgnode nil))))))))))

;;; last mod 4/30/92 based on Takahashi (TIS) to be same as COLING92
(defun copy-node-comp-forwarded (dgnode)
  "When the node to be copied is a result of forwarding then,
   we will need to record changes.
   We will not need to copy atomic and leaf nodes at all."
  (declare (type dgnode dgnode))
  (cond ((ATOMICNODE-p dgnode)
 (values dgnode :changed)) ;;; considered change
((LEAFNODE-p dgnode)
 (values dgnode :changed))
(T   ;;; complex-dgnode
 (cond ((and (DGNODE-comp-arc-list dgnode)
     (= *unify-global-counter*
(DGNODE-generation dgnode)))
(let ((newdgnode (%create-dgnode)))
  (declare (type dgnode newdgnode))
  (setf
   (DGNODE-type newdgnode) :complex
                   (DGNODE-generation newdgnode) *unify-global-counter*
   ;; (DGNODE-name newdgnode) (DGNODE-name dgnode)
   (DGNODE-copy dgnode) newdgnode
   )
  (dolist (comp-arc (DGNODE-comp-arc-list dgnode))
                     (declare (type dgarc comp-arc))
    (push (copy-arc-and-comp-arc-share comp-arc)
  (DGNODE-arc-list newdgnode)))
  (dolist (arc (DGNODE-arc-list dgnode))
                     (declare (type dgarc arc))
    (push (copy-arc-and-comp-arc-share arc)
  (DGNODE-arc-list newdgnode)))
  (setf (DGNODE-comp-arc-list dgnode) nil)  ;;; 10/1/91 for GC.
  (values newdgnode :changed)))
                 (t (let ((state nil))
                     (declare (type symbol state))
                     (setf (DGNODE-copy dgnode) dgnode
                           (DGNODE-generation dgnode) *unify-global-counter*)
                     ;;; --- This hack is needed to avoid infinite loop
                     ;;;  with a cyclic graph.  By making a copy to be
                     ;;;  itself infinite loop can be avoided. 10/15/91 tomabech
                     (let ((arcs (map-dolist (arc (DGNODE-arc-list dgnode))
                             (multiple-value-bind (arc changed)
                                      (copy-arc-and-comp-arc-share arc)
                                     (declare (type dgarc arc)
                                              (type symbol changed))
                                     (if changed
                                         (setq state changed))
                             arc))))
                           (cond (state
                             (cond ((not (eq (DGNODE-copy dgnode) dgnode))
                                     (setf
                                      (DGNODE-arc-list (DGNODE-copy dgnode)) arcs
                                      (DGNODE-type (DGNODE-copy dgnode)) :complex
                                      )
                                     (values (DGNODE-copy dgnode) :changed))
                                    (t
                                     (let ((newdgnode (%create-dgnode)))
                                       (declare (type dgnode newdgnode))
                                       (setf
                                        (DGNODE-type newdgnode) :complex
                                        (DGNODE-generation newdgnode) *unify-global-counter*
```

```
                              ;;;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
                                     (DGNODE-arc-list newdgnode) arcs
                                     (DGNODE-copy dgnode) newdgnode
                                     )
                                    (values newdgnode :changed)))))
                            (t (setf (DGNODE-copy dgnode) nil)
                            ;;; This corresponds to the above hack.
                            ;;; Reset the copy field when actually no
                            ;;; copy was made. 10/15/91 tomabech
                            (values dgnode :changed))))))))))

;;; structure sharing supported.  Last mod 10/2/91,4/30/92,5/6/92 tomabech
;;; Whether the node is forwarded is checked by 'eq' between dgnode-underef and
;;; dgnode.  This should be same as actually checking the currency of the forward
;;; field and is faster.  If this causes any bug, use the commented code above.
(defun copy-dg-with-comp-arcs-share (dgnode-underef)
  "We don't copy atomic nodes since their values are constant.  Leaf nodes
   need not be copied either since, if they unify with something they get
   forwarded. Thus, if a dereferencing operation returns a leaf or the
   original node is the leaf, it can simply remain that way.
   Only make a copy only either when any of my subnodes created copy
   or I have a valid comp-arc-list or temporary forwarding.
   Recursively go down the dgnode and make a copy of the dg.
   Content of the comp-arc-list in the original dg should be
   put in the arc-list of the copy. Ignore the comp-arc-list with
   old generation stamp.
   If the node to be copied is a result of dereferencing then copy must
   be made. Otherwise, copy is made only when any of the nodes below are
   copied (changed)."
  (declare (type dgnode dgnode-underef))
  (let ((dgnode (dereference-dg dgnode-underef)))
   (declare (type dgnode dgnode))
   (cond ((and (DGNODE-copy dgnode)              ;;; if a current copy exists
           (= (DGNODE-generation (DGNODE-copy dgnode)) *unify-global-counter*))
              (if (eq dgnode (DGNODE-copy dgnode))
                (let ((newdgnode (%create-dgnode)))
                  (declare (type dgnode newdgnode))
                  (setf
                   (DGNODE-type newdgnode) :bottom
                   (DGNODE-generation newdgnode) *unify-global-counter*
                   ;; (DGNODE-name newdgnode) (DGNODE-name dgnode)
                   (DGNODE-copy dgnode) newdgnode
                   )
                  (values newdgnode :changed))
                (values (DGNODE-copy dgnode) :changed)))
  ((eq dgnode dgnode-underef)
   (copy-node-comp-not-forwarded dgnode))
  (t (copy-node-comp-forwarded dgnode)))))

;;; original
(defun simple-copy-dg-no-share (dgnode-underef)
  "Recursively go down the dgnode and make a copy of the dg.
   This one is used by external functions requiring copying of dg.
   the content of comp-arc-list is not respected."
  (declare (type dgnode dgnode-underef))
```

```
 (let ((dgnode (dereference-dg dgnode-underef)))
  (declare (type dgnode dgnode))
  (cond ((and (DGNODE-copy dgnode)               ;;; if a current copy exists
       (= (DGNODE-generation (DGNODE-copy dgnode)) *unify-global-counter*))
 (DGNODE-copy dgnode))                    ;;; then return the copy
        ((ATOMICNODE-p dgnode)
 (let ((newdgnode (%create-dgnode)))
   (declare (type dgnode newdgnode))
   (setf
     (DGNODE-type newdgnode) :atomic
 ;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
     (DGNODE-arc-list newdgnode) (DGNODE-arc-list dgnode)
     (DGNODE-generation newdgnode) *unify-global-counter*
     (DGNODE-copy dgnode) newdgnode
     )
   newdgnode))
((LEAFNODE-p dgnode)
 (let ((newdgnode (%create-dgnode)))
   (declare (type dgnode newdgnode))
   (setf
     (DGNODE-type newdgnode) :leaf
 ;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
     (DGNODE-generation newdgnode) *unify-global-counter*
     (DGNODE-copy dgnode) newdgnode
     )
   newdgnode))
(T  ;;;  complex-dgnode
 (let ((newdgnode (%create-dgnode)))
   (declare (type dgnode newdgnode))
   (setf
     (DGNODE-type newdgnode) :complex
 ;;      (DGNODE-name newdgnode) (DGNODE-name dgnode)
     (DGNODE-generation newdgnode) *unify-global-counter*
     (DGNODE-copy dgnode) newdgnode   ;;; moved from after recursion 10/15/91
     )
   (dolist (arc (DGNODE-arc-list dgnode))
             (declare (type dgarc arc))
     (push (simple-copy-arc arc)
       (DGNODE-arc-list newdgnode)))
   newdgnode)))))

(defmacro set-forward-dgnode (dgnode1 dgnode2)
  (declare (type dgnode dgnode1 dgnode2))
  "This is an interface for Kogure/Kato earley-based parser"
   '(forward-dg ,dgnode1 ,dgnode2 :permanent))


;;; END OF TOMABECHI'S UNIFICATION ALOGRITHM DEFINITION
;;;==========================================================

;;; Interface for Parser for Data gathering 10/17/91 tomabech

(defun tana (sent)
  (declare (special *dgnodes* *dgarcs* *unify0 *unify1*
    *NUMBER-OF-UNIFY-SUCCESS*
    *NUMBER-OF-UNIFY-FAIL*))
  (setq *dgnodes* 0)
  (setq *dgarcs* 0)
  (setq *unify0* 0)
  (setq *unify1* 0)
  (setq *NUMBER-OF-UNIFY-SUCCESS* 0)
  (setq *NUMBER-OF-UNIFY-FAIL* 0)
  (let ((start-time (get-internal-real-time))
(time-spent 0.0))
    (ana sent)
    (setq time-spent (- (get-internal-real-time) start-time))
    (format t "~% Number of Nodes Created:  ~A" *dgnodes*)
    (format t "~% Number of Arcs  Created:  ~A" *dgarcs*)
    (format t "~% Number of UNIFY0s Called: ~A" *unify0*)
    (format t "~% Number of UNIFY1s Called: ~A" *unify1*)
    (format t "~% Unification Success Rate: ~A" (usrate))
    (display-time time-spent)
    (terpri)
    ))

 (defun display-time (time)
   (let ((small 0.0)
 (big 0.0))
   (multiple-value-setq (big small)
     (floor time internal-time-units-per-second))
   (format t "~% Took ~D.~D seconds of real time."
     big small)))


;;;=====================================================================
;;; The code below is written by Dr. Kiyoshi Kogure of NTT (and ATR)
;;; We standardizedly use his node print functions and therefore, it is
;;; provided here under his permission.
;;;=====================================================================
;;; DGNODE PRINT FUNCTIONS
;;; Pretty Printing graphs, originally defined by Kiyoshi Kogure
;;; some modifications by tomabech

;;;(defun kprint-dgnode (x stream plevel)
;;; "Kogure type dgnode print function. Useful for debugging"
;;;        (declare (ignore plevel))
;;;        (setq x (dereference-dg x))
;;;        (format stream "~s{~s}" (dgnode-type x)
;;;                (cond ((complex-dgnode-p x)
;;;       (map 'list #'arc-label (dgnode-arc-list x)))
;;;                      (t (dgnode-arc-list x)))))

 (defmacro pprint-fs-get-dgnode-name (dgnode)
   '(second (assoc ,dgnode *dgnode-assigns*)) )

 (defmacro pprint-fs-make-dgnode-name ()
   '(format nil "X~2,VD" #\0 (incf *dgnode-counter*)) )

 (defmacro pprint-fs-force-get-dgnode-name (dgnode)
   (let ((name (gensym)))
  `(let ((,name (pprint-fs-get-dgnode-name ,dgnode)))
    (cond ((null ,name)
  (push (list ,dgnode (setq ,name (pprint-fs-make-dgnode-name))) *dgnode-assigns*)
  ,name)
 (t
  ,name))) ))

;;;
;;;     Pretty-Printing a Feature Structure
;;;
(defun pprint-fs (fs &optional (stream *debug-stream1*)
     &key (init-indent 0) (indent-step 1) (return-p t))
  (let ((*dgnode-assigns* nil)
(*dgnode-counter* 0))
    (declare (special *dgnode-assigns* *dgnode-counter*))
    (when return-p (format stream "~%"))
;   (reorder-feature fs)
    (pprint-fs-internal fs stream init-indent indent-step)) )

(defun pprint-dg (fs stream plevel) ;; fs is a dg
 "Same as pprint-fs. This is modified to be used as a print function
  for defstruct. Takes three arguments: structure, stream, and level."
 (declare (ignore plevel))
 (if (not (DGNODE-p fs))
     (error "~% ERROR in pprint-dg, ~a not graph-structure." fs))
 (let ((init-indent 0)
(indent-step 1)
(return-p t)
(*dgnode-assigns* nil)
(*dgnode-counter* 0))
    (declare (special *dgnode-assigns* *dgnode-counter*))
    (when return-p (format stream "~%"))
;   (reorder-feature fs)
    (pprint-fs-internal fs stream init-indent indent-step)) )

(defun find-real-result-dgnode (dgnode)
 "modified on 3/19/91 adding dereference operation"
 (dereference-dg dgnode))

(defun pprint-fs-leaf (nil1 stream init-indent nil2 name)
  (declare (ignore nil1 nil2))
  (format stream "~VT~A[]" init-indent name) )

(defun pprint-fs-atomic (fs stream init-indent ignore name)
  (declare (ignore ignore))
  (format stream "~VT~A ~A" init-indent name (DGNODE-arc-list fs)) )

(defun pprint-fs-complex (fs stream init-indent indent-step name)
  (let ((string (format nil "~VT~A[" indent-step name))
(arcs   (dgnode-arc-list fs)))
    (format stream "~A" string)
    (setq init-indent (+ init-indent (length string)))
    (pprint-fs-arc (first arcs) stream init-indent indent-step)
    (mapc #'(lambda (arc)
      (format stream "~%~VT" init-indent)
      (pprint-fs-arc arc stream init-indent indent-step))
   (rest arcs))) )

(defun pprint-fs-internal2 (fs stream init-indent indent-step &aux name)
  (setq name (pprint-fs-force-get-dgnode-name fs))
  (case (dgnode-type-of fs)
    (:leaf    (pprint-fs-leaf    fs stream init-indent indent-step name))
    (:atomic  (pprint-fs-atomic  fs stream init-indent indent-step name))
    (:complex (pprint-fs-complex fs stream init-indent indent-step name))) )

(defun pprint-fs-internal (fs stream init-indent indent-step &aux name)
  (if (not (DGNODE-p fs))
      (error "~% ERROR1 in pprint-fs-internal, ~a not graph-structure." fs))
  (setf fs (find-real-result-dgnode fs))
  (if (not (DGNODE-p fs))
      (error "~% ERROR2 in pprint-fs-internal, ~a not graph-structure." fs))
  (if (not (null (setq name (pprint-fs-get-dgnode-name fs))))
      (format stream "~VT~A" init-indent name)
      (pprint-fs-internal2 fs stream init-indent indent-step)))

;;; mod. 3/23/91
(defun pprint-fs-arc (arc stream init-indent indent-step)
 (if (not (ARC-p arc))
     (error "~% ERROR in pprint-fs-arc, ~a not arc-structure." arc))
  (let ((string (format nil "[~A~VT" (arc-label arc) indent-step)))
    (format stream "~A" string)
    (pprint-fs-internal (arc-value arc) stream
(+ init-indent (length string)) indent-step)
    (format stream "]")))

;;; The end of Kogure's node print functions.
;;;-------------------------------------------------------------------------

;;==========================================================================
(terpri)
(print "   Quasi-destructive Graph Unification Package -- Version 5.3      ")
(print "         Copyright (C) 1990, 1993  by  Hideto Tomabechi.           ")
(print "                         All rights reserved.                      ")
(print " This code is put in public domain. Any modifications, suggestions,")
(print " and bug reports should be addressed to tomabech@cs.cmu.edu, or    ")
(print " tomabech@is.tokushima-u.ac.jp, or tomabech@mtlab.sfc.keio.ac.jp   ")
(print "                                                                   ")
(terpri)
(print "Do (setq *inheritance* t) to enable HyperFrame inheritance support.")
(print "   (setq *str-sharing* nil) to disable structure sharing scheme.   ")
(print "   (tana sent1) to parse first sentence.                           ")
;;==========================================================================
;;; eof
```

# Appendix V: Sample Parser Output

Below is provided simply to show the idea of the kind of grammar used in the experiments.

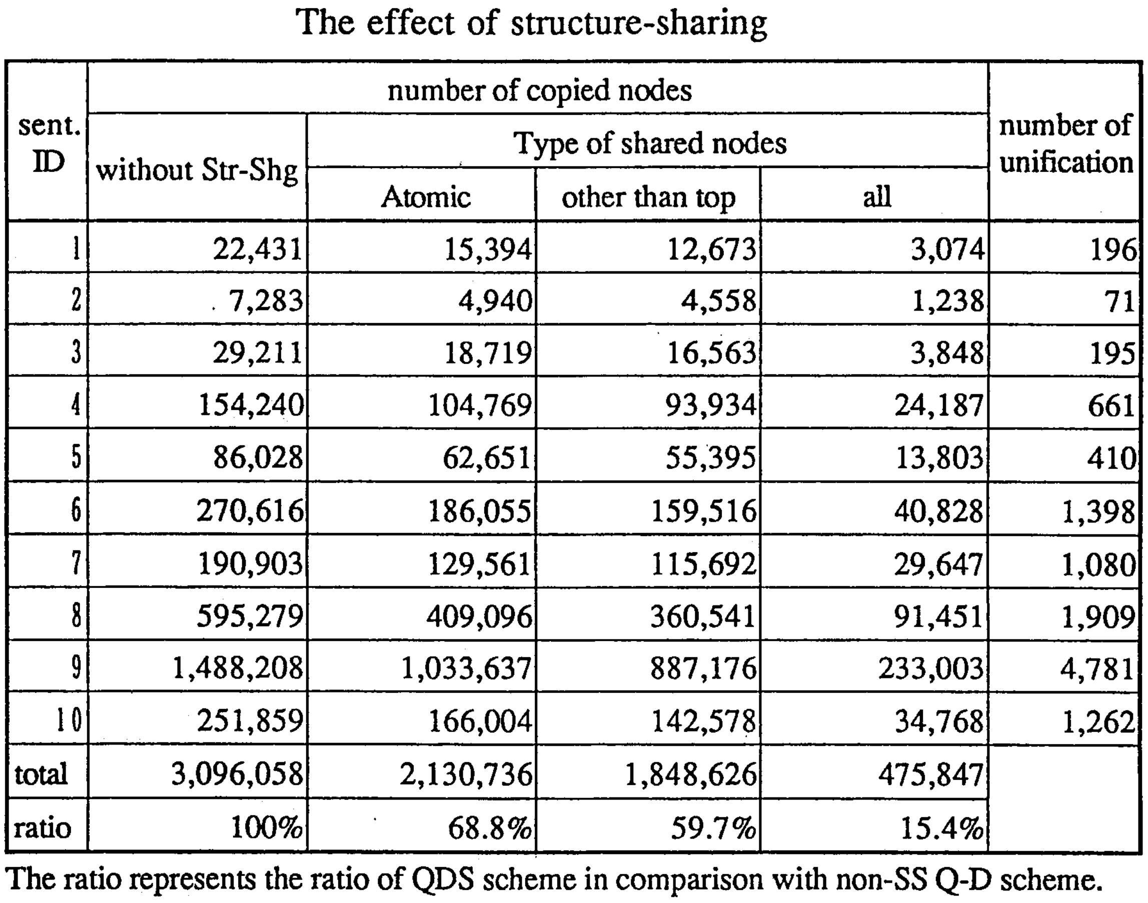
This output is on a Sun/Sparc2 station with a Lucid CommonLisp.

```
> (tana sent1)
INPUT SENTENCE:MOSHIMOSHI
 X01[[PRAG  X02[[HEARER X03[]]
                [SPEAKER X04[]]]
     [SEM  X05[[RECP X03]
               [AGEN X04]
               [RELN X06 MOSHIMOSHI-HELLO]]
     [SUBCAT X07 END]
     [HEAD  X08[[CFORM X09 SENF]
                [CTYPE X10 NONC]
                [POS X11 V]]
 Number of Nodes Created:  79
 Number of Arcs  Created:  123
 Number of UNIFY0s Called: 6
 Number of UNIFY1s Called: 27
 Unification Success Rate: 0.5
 Took 0.178746 seconds of real time.
NIL
> (tana sent9)
INPUT SENTENCE:SOREDEHAKOCHIRAKARASOCHIRANITOUROKUYOUSHIWOOOKURIITASHIMASU
 X01[[PRAG  X02[[RESTR  X03[[REST  X04[[REST  X05[[REST X06[]]
                                                  [FIRST  X07[[FIRST  X08[[REST X09[]]
                                                                          [FIRST  X10[[FIRST X11[]]
                                                                                      [REST X12[]]]]
                                                              [REST X13[]]]]
                                       [FIRST  X14[[FIRST  X15[[RELN X16 POLITE]
                                                               [AGEN  X17[[LABEL X18 *SPEAKER*]]
                                                               [RECP  X19[[LABEL X20 *HEARER*]]]
                                                   [REST  X21[[FIRST  X22[[RELN X23 RESPECT]
                                                                          [AGEN X17]
                                                                          [RECP X19]]
                                                              [REST X24 END]]]]
                            [FIRST  X25[[FIRST  X26[[RELN X27 POLITE]
                                                    [AGEN X17]
                                                    [RECP X19]]
                                        [REST X28 END]]]
                [HEARER X19]
                [SPEAKER X17]]
     [SEM  X29[[INFMANN  X30[[RESTR  X31[[OBJE X32[]]
                                         [RELN X33 SOREDEHA-1]]
                             [PARM X32]]
               [RECP X19]
               [AGEN X17]
               [OBJE  X34[[PARM X35[]]
                          [RESTR  X36[[RELN X37 TOUROKUYOUSHI-1]
```

```
                                      [OBJE X35]]]
               [RELN X38 OKURU-1]]
     [SLASH X39[]]
     [SUBCAT X40 END]
     [HEAD  X41[[POS X42 V]
                [CTYPE X43 MASU]
                [CFORM X44 SENF]
                [MODL  X45[[POLT X46 +]]]
 Number of Nodes Created:  9161
 Number of Arcs  Created:  12666
 Number of UNIFY0s Called: 488
 Number of UNIFY1s Called: 3373
 Unification Success Rate: 0.3668032786885246
 Took 4.966781 seconds of real time.
NIL
> (tana sent12)
INPUT SENTENCE:KOCHIRAKARASOCHIRANITOUROKUYOUSHIWOSHIKYUUNIOOKURIITASHIMASU
 X01[[PRAG  X02[[RESTR  X03[[REST  X04[[REST  X05[[REST X06[]]
                                                  [FIRST  X07[[FIRST  X08[[REST X09[]]
                                                                          [FIRST  X10[[FIRST X11[]]
                                                                                      [REST X12
                                                              [REST X13[]]]]
                                       [FIRST  X14[[FIRST  X15[[RELN X16 POLITE]
                                                               [AGEN  X17[[LABEL X18 *SPEAKER*]]
                                                               [RECP  X19[[LABEL X20 *HEARER*]]]
                                                   [REST  X21[[FIRST  X22[[RELN X23 RESPECT]
                                                                          [AGEN X17]
                                                                          [RECP X19]]
                                                              [REST X24 END]]]]
                            [FIRST  X25[[FIRST  X26[[RELN X27 POLITE]
                                                    [AGEN X17]
                                                    [RECP X19]]
                                        [REST X28 END]]]
                [HEARER X19]
                [SPEAKER X17]]
     [SEM  X29[[RECP X19]
               [AGEN X17]
               [OBJE  X30[[RESTR  X31[[OBJE X32[]]
                                      [RELN X33 TOUROKUYOUSHI-1]]
                          [PARM X32]]
               [RELN X34 OKURU-1]
               [MANN  X35[[PARM X36[]]
                          [RESTR  X37[[RELN X38 SHIKYUUNI-1]
                                      [OBJE X36]]]]
     [SLASH X39[]]
     [SUBCAT X40 END]
     [HEAD  X41[[POS X42 V]
                [CTYPE X43 MASU]
                [CFORM X44 SENF]
                [MODL  X45[[POLT X46 +]]]
 Number of Nodes Created:  7690
 Number of Arcs  Created:  10485
 Number of UNIFY0s Called: 436
 Number of UNIFY1s Called: 3126
 Unification Success Rate: 0.3348623853211009
 Took 3.162755 seconds of real time.
NIL
>
```

# Appendix VI: An External Empirical Result

Table 7.1 was taken from Takahashi *et al*[1992]. It is one of the first sets of data that came out of the ATR's large scale speech-to-speech translation project (ASURA). The project team has adopted the Q-D and QDS method and has been conducting some interesting experiments using their large scale grammar. The Q-D and QDS algorithms in the ASURA project uses Kasper's method for disjunctive feature structures and Kogure's method for negative feature structure. As we can see from the data below, the QDS method reduced the number of copies from the Q-D method significantly. More data should be available from the project during 1993.



The effect of structure-sharing

| sent. ID | number of copied nodes | | | | number of unification |
|---|---|---|---|---|---|
| | without Str-Shg | Type of shared nodes | | | |
| | | Atomic | other than top | all | |
| 1 | 22,431 | 15,394 | 12,673 | 3,074 | 196 |
| 2 | 7,283 | 4,940 | 4,558 | 1,238 | 71 |
| 3 | 29,211 | 18,719 | 16,563 | 3,848 | 195 |
| 4 | 154,240 | 104,769 | 93,934 | 24,187 | 661 |
| 5 | 86,028 | 62,651 | 55,395 | 13,803 | 410 |
| 6 | 270,616 | 186,055 | 159,516 | 40,828 | 1,398 |
| 7 | 190,903 | 129,561 | 115,692 | 29,647 | 1,080 |
| 8 | 595,279 | 409,096 | 360,541 | 91,451 | 1,909 |
| 9 | 1,488,208 | 1,033,637 | 887,176 | 233,003 | 4,781 |
| 10 | 251,859 | 166,004 | 142,578 | 34,768 | 1,262 |
| total | 3,096,058 | 2,130,736 | 1,848,626 | 475,847 | |
| ratio | 100% | 68.8% | 59.7% | 15.4% | |

The ratio represents the ratio of QDS scheme in comparison with non-SS Q-D scheme.

Table 7.1: The Effect of Structure Sharing.