Internal Use Only (非公

後に論文公開済み

TR-I-0220

データ構造共有型複製方式準破壊型単一化手法

― 超並列制約伝播プロジェクトで開発されたアルゴリズムの単一化文法パーザーでの使用実験 ―

Quasi-Destructive Graph Unification with Structure-Sharing

Hideto Tomabechi *

1991.11

*Visiting Research Scientist from Carnegie Mellon University

Abstract

Graph unification remains the most expensive part of unification-based natural language constraint processing. We focus on one speed-up element in the design of unification algorithms: avoidance of copying of unmodified subgraphs. We propose a method of attaining such a design through a method of structure-sharing which avoids log(d) overheads often associated with structure-sharing of graphs, without any use of costly dependency pointers. The proposed scheme eliminates *redundant copying* while maintaining the quasi-destructive scheme's ability to avoid *over copying* and *early copying* combined with its ability to handle cyclic structures without algorithmic additions. This algorithm was originally developed by the author as a fast graph-unification method in the joint ATR/CMU massively-parallel constraint propagation natural language processing project and is now used in the conventional unification-based grammar parsing at ATR and at CMU.

グラフの単一化は単一化を利用した自然言語処理において計算時間のボトルネックになっている。本稿では準破壊型単一化にデータ構造レベルでの表現内容の共有化を利用したグラフの複製手法を導入する方法について述べる。この単一化手法は超並列制約伝播解析における制約処理手法として当初開発されたが、ループを扱える一般化された単一化手法であり、通常の単一化文法パーザー等で利用可能であり、HPSG ベースの日本語文法のアーレーのアルゴリズムを利用した解析では Wroblewski の手法の 3 倍から 5 倍程度の実行速度を確認した。

ATR 自動翻訳電話研究所

ATR Interpreting Telephony Research Laboratories

© (株) ATR 自動翻訳電話研究所 1991

©1991 by ATR Interpreting Telephony Research Laboratories

Contents

1 Q-D Method with Structure-Sharing **1**

1.1 Motivation 1
1.2 Quasi-Destructive Graph Unification 3
1.3 Q-D Copying with Structure-Sharing 10
1.4 Experiments 14
1.5 Discussion: 16
1.6 Conclusion 19

2 破壊型単一化から準破壊型単一化へ **22**

2.1 自然言語処理におけるグラフの単一化の役割 22
2.2 グラフ単一化の問題点 - 計算量（実行速度） 22
2.3 高速なグラフ単一化手法に向けた課題 23
2.4 Pereira の手法 23
2.5 Kartunnen の手法 24
2.6 Wroblewski の手法 24
2.7 高速な汎用グラフ単一化アルゴリズムの提案 25
2.8 高速化問題の切口 25
2.9 単一化の失敗 25
2.10 複製行為の負荷 26
2.11 無変更のグラフの共有化 26
2.12 解決への方向 26
2.13 Over-Copying と Early-Copying 26
2.14 進行型複製手法（Incremental Copying）の本質的問題 27
2.15 基本的手法 27
2.16 ループあるいはリエントランシーと一時的フォワード 27
2.17 時間差準破壊型手法による単一化例 29
2.18 時間差準破壊型アルゴリズムの単一化における特徴 30
2.19 グラフ単一化におけるデータ構造の共有： 定義 30

2.20 小暮の手法 （Coling'90） 30
2.21 Emele の手法 （ACL'91） 31
2.22 時間差準破壊型手法における構造の共有 − 基本的な考え方 32
2.23 時間差準破壊型手法における構造の共有 − アルゴリズムの変更 . 32
2.24 構造共有化 （Data-Structure Sharing）の一つの問題点 33
2.25 Bottom 共有による問題の解決法：（いくつかの手法例） 33
2.26 結論 33

Chapter 1

Q-D Method with Structure-Sharing

1.1 Motivation

Despite recent efforts in improving graph unification algorithms, graph unification remains the most expensive part of parsing, both in time and space. ATR's latest data from the SL-TRANS large-scale speech-to-speech translation project ([Morimoto, *et al*, 1990]) show 80 to 90 percent of total parsing time is still consumed by graph unification where 75 to 95 percent of time is consumed by graph copying functions.[1] Quasi-Destructive (Q-D) Graph Unification ([Tomabechi, 1991]) was developed as a fast variation of non-destructive graph unification based upon the notion of time-sensitive 'quasi-destruction' of node structures. The Q-D algorithm was proposed based upon the following accepted observation about graph unification:

Unification does not always succeed.

Copying is an expensive operation.

The design of the Q-D scheme was motivated by the following two principles for fast graph unification based upon the above observations:

- **Copying should be performed only for successful unifications.**

[1] Based on unpublished reports from Knowledge and Data Processing Dept, ATR. The observed tendency was that sentences with very long parsing time requiring a large number of unification calls (over 2000 top-level calls) consumed extremely large proportion (over 93 percent) of total parsing time for graph unification. Similar data reported in [Kogure, 1990].

- **Unification failures should be found as soon as possible.**

and eliminated Over Copying and Early Copying (as defined in [Tomabechi, 1991][2]) and ran about twice the speed of [Wroblewski, 1987]'s algorithm.[3] In this paper we propose another design principle for graph unification based upon yet another accepted observation that:

Unmodified subgraphs can be shared.

At least two schemes have been proposed recently based upon this observation (namely [Kogure, 1990] and [Emele, 1991]); however, both schemes are based upon the incremental copying scheme and as described in [Tomabechi, 1991] incremental copying schemes inherently suffer from *Early Copying* as defined in that article. 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.*

---

[2]Namely,

- **Over Copying:** Two dags are created in order to create one new dag. – This typically happens when copies of two input dags are created prior to a destructive unification operation to build one new dag.
- **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 defined Early Copying as follows: "The argument dags are copied *before* unification started. If the unification fails then some of the copying is wasted effort" and restricts early copying to cases that only apply to copies that are created prior to a unification. Our definition of Early Copying includes copies that are created during a unification and created up to the point of failure which were uncovered by Wroblewski's definition.

[3]Recent experiments conducted in the Knowledge and Data Processing Dept. of ATR shows the algorithm consistently runs at about 40 percent of the elapsed time of Wroblewski's algorithm with its SL-TRANS large-scale spoken-language translation system (with over 10,000 grammatical graph nodes).

Virtually all graph-unification algorithms support FS-Sharing and some support DS-Sharing with varying levels of overhead. In this paper 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 paper, 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 Bottom nodes can be shared**[4] – Atomic nodes can be shared safely since they never change their values. Bottom nodes can be shared[5] since bottom 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 quasi-destructive scheme).

By designing an algorithm based upon these principles for structure-sharing while retaining the quasi-destructive nature of [Tomabechi, 1991]'s algorithm, our scheme eliminates Redundant Copying while eliminating both Early Copying and Over Copying.

1.2 Quasi-Destructive Graph Unification

We would first like to describe the quasi-destructive (Q-D) graph unification scheme which is the basis of our scheme. As a data structure, a node is represented with five fields: type, arc-list, comp-arc-list, forward, copy, and generation.[6] 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.

[4] Atomic nodes are nodes that represent atomic values, Bottom nodes are nodes that represent variables.

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

[6] Note that [Tomabechi, 1991] used separate mark fields for 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.

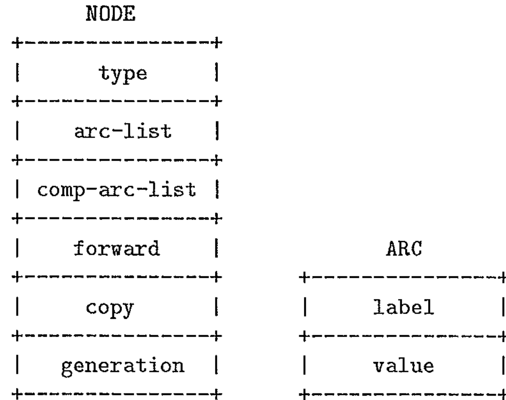


Figure 1.1: **Node and Arc Structures**

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 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 only valid during one top-level graph 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 only valid 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[7]. As in [Pereira, 1985], the Q-D algorithm has three types of nodes: 1) :atomic,

[7] 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 whenever the generation mark is not 9, the integer value must equal the global counter value to respect the forwarding link.

2) :bottom[8], and 3) :complex. :atomic type nodes represent atomic symbol values (such as 'Noun'), :bottom 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. :bottom nodes succeed in unifying with any nodes and the result of unification takes the type and the value of the node that the :bottom node was unified with. :atomic nodes succeed in unifying with :bottom nodes or :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 :bottom nodes or with :complex nodes whose subgraphs all unify.[9] Figure 2 is the central quasi-destructive graph unification algorithm and Figure 3 is the dereferencing[10] function. Figure 4 shows the algorithm for copying nodes and arcs (called from unify0) while respecting the contents of comp-arc-lists.

[8] Bottom is called leaf in Pereira's algorithm.

[9] Arc values are always nodes and never symbolic values because :atomic and :bottom nodes may be (or become) pointed to by multiple arcs (i.e, FS-Sharing) depending on grammar constraints, and we do not want arcs to contain terminal atomic values.

[10] Dereferencing is an operation to recursively traverse forwarding links to return the target node of forwarding.

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 11
 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 = dg2)12THEN
    return('*T*);
 ELSE IF (dg1.type = :bottom) THEN
    forward-dg(dg1,dg2,:temporary);
    return('*T*);
 ELSE IF (dg2.type = :bottom) 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)13THEN
              forward-dg(dg2,dg1,:temporary);
              return('*T*);
           ELSE throw14with 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);15
      new ← complementarcs(dg2,dg1);16
      IF17(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;
```

Figure 2: **The Q-D Unification Functions**

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
         dereference-dg(forward-dest);
     ELSE dg.forward ← nil; ;; make it GCable
           return(dg);
 ELSE return(dg);
END;
```

Figure 3: **The Q-D Dereference Function**

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 matchs the current value of the global timing counter. Forward(dg1, dg2, :forward-type)

[11] 9 indicates a permanent forwarding link.

[12] Equal in the 'eq' sense. Because of forwarding and cycles, it is possible that dg1 and dg2 are 'eq'.

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

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

[15] 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*.

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

[17] This check was added after [Tomabechi, 1991] 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.

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.[18] 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 dg2 if dg1 is of type :bottom). 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.[19]

[18] 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.

[19] 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.

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[20]= *unify-global-counter*) THEN
      return(dg.copy);[21]
 ELSE IF (dg.type = :atomic) THEN
      newcopy ← create-node();[22]
      newcopy.type ← :atomic;
      newcopy.arc-list ← dg.arc-list;
      newcopy.generation ← *unify-global-counter*;
      dg.copy ← newcopy;
      return(newcopy);
 ELSE IF (dg.type = :bottom) THEN
      newcopy ← create-node();
      newcopy.type ← :bottom;
      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;[23]
      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;

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;
```

Figure 4: **Node and Arc Copying Functions**

1.3 Q-D Copying 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 structure-sharing 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 is desired (by simply choosing different copying functions). Informally, the Q-D copying with structure-sharing is performed in the following way. Atomic and bottom 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 multiple-value binding facility when a copy of the nodes are recursively returned. Two values are returned, the first value being the copy (or original) node and the second value being the flag representing whether any of the node below that node (including that node) has been changed. Atomic and bottom nodes are always shared; however, they are considered changed if they were a target of forwarding so that the 'changed' information is passed up. If the complex node is a target of forwarding, 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 the Q-D copying with structure-sharing.[24]

---

[20]I.e., the 'generation' field of the node stored in the 'copy' field of the 'dg' node. The algorithm described in [Tomabechi, 1991] 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, 1991] 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.

[24]One thing to be noted is that when a complex node is a point of convergence, and when there was no change in the subgraph, the provided algorithm will perform the unnecessary recursion for the second traversal through the arc pointing to the convergent node. Since there is no change in the complex graph, this will not change the result in anyway, but if the complex graph is very large it could be costly. If there was a change in the complex graph this will not happen since the convergent node will be copied in the first traversal of the

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);[25]
 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 = :bottom) 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[26], dg.generation ← *unify-global-counter*;
        FOR ALL arc IN dg.arc-list DO
          newarc,changed ← copy-arc-and-comp-arc-share(arc); [27]
          push newarc into arcs;
          IF (changed has value) THEN
              state ← changed;
```

convergent arc and that copy is simply returned in the second traversal of the convergent arc. One way to avoid it is to local write :changed or :unchaged information into the :copy field of the node and immediately return the input node with either :changed or :unchaged flag, so that such information will be written in the first traversal of the convergent arc and utilized in the second traversal of the convergent arc. This was suggested by Masaaki Nagata of ATR and has been included in their implementation of the algorithm.

```
    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 = :bottom) 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;
```

Figure 5: **Structure-Sharing Copying Functions**

1.4 Experiments

Table 1 shows the results of our experiments using an HPSG-based sample Japanese grammar developed at ATR for a conference registration telephone dialogue domain. 'Unifs' 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')[28]. 'USrate' represents the ratio of successful unifications to the total number of unifications. We parsed each sentence three times on a Symbolics 3620 using three unification methods, namely, Wroblewski's algorithm, a quasi-destructive method without structure-sharing, and a quasi-destructive method with structure-sharing. We took the shortest elapsed time for each method ('W' represents Wroblewski's algorithm with a modification to handle cycles and variables[29], 'QD' represents the quasi-destructive method without structure-sharing, and 'QS' represents the proposed method with structure-sharing). Data structures are the same for all three unification methods except for additional fields for comp-arc-list in the Q-D methods. Same functions are used to interface with Earley's parser and the same subfunctions are used wherever possible (such as creation and

[23]'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.

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

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

[28]Unify1 is called several times the number of unify-dg in the grammar used in the experiment. For example unify1 was called 3299 times for sentence 9 when unify-dg was called 480 times.

[29]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. And 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. Handling variables in Wroblewski's algorithm is basically the same as in our algorithm (i.e., Pereira's scheme), and the addition of this functionality can be ignored in terms of comparison to our algorithm. Our algorithm does not require any additional scheme to handle cycles in input dgs.

access of arcs) to minimize the differences that are not purely algorithmic. 'Number of Copies' represents the number of nodes created during each parse. 'Number of Arcs' represents the number of arcs created during each parse.

sent#	Unifs	USrate	Elapsed time(sec)			Num of Copies			Num of Arcs		
			W	QD	QS	W	QD	QS	W	QD	QS
1	6	0.50	0.20	0.15	0.13	107	79	18	113	123	36
2	101	0.34	2.53	1.16	1.10	2285	1317	407	2441	1917	760
3	18	0.22	0.40	0.20	0.20	220	111	26	182	183	62
4	71	0.55	2.20	1.24	0.91	2151	1564	514	2408	2191	879
5	305	0.37	13.78	6.51	3.65	9092	5224	1220	9373	7142	2272
6	59	0.27	3.20	0.64	0.50	997	549	97	874	797	204
7	6	0.50	0.21	0.13	0.11	107	79	18	113	123	36
8	81	0.51	3.17	1.59	1.21	2406	1699	401	2572	2334	710
9	480	0.37	24.62	8.11	5.74	15756	8986	1696	17358	12427	3394
10	555	0.41	40.15	16.39	8.80	18822	11234	2737	20323	15375	5116
11	109	0.45	4.60	1.71	1.41	2913	1938	555	3089	2712	992
12	428	0.33	19.57	8.24	4.45	13363	7491	1586	14321	10218	3059
13	559	0.39	37.76	11.74	6.23	17741	9417	2483	19014	13055	4471
14	52	0.38	3.81	0.90	0.50	947	693	107	893	983	199
15	77	0.55	2.50	1.57	0.93	2137	1513	428	2436	2185	793
16	77	0.55	2.53	1.57	0.90	2137	1513	428	2436	2185	793
total	2984		161.23	61.85	36.77	91181	53407	12721	97946	73950	23776
(% for total)			100%	38.4%	22.8%	100%	58.6%	14%	100%	76%	24%

Table 1.1: **Comparison of three methods**

We used Earley's parsing algorithm for the experiment. The Japanese grammar is based on HPSG analysis ([Pollard and Sag, 1987]) covering phenomena such as 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 grammar covers many of the important linguistic phenomena in conversational Japanese. The grammar graphs which are converted from the path equations contain 2324 nodes.[30] We used 16 sentences from a sample telephone conversation dialog which range from very short sentences (one word,

[30] Disjunctive equations are preprocessed by the grammar reader module to expand into cross-multiples, whereas in ATR's SL-TRANS system, Kasper's method ([Kasper, 1987]) to handle disjunctive feature-structures is adopted.

i.e., *iie* 'no') to relatively long ones (such as *soredehakochirakarasochirani-tourokuyoushiwoookuriitashimasu* 'In that case, we [speaker] will send you [hearer] the registration form.'). Thus, the number of (top-level) unifications per sentence varied widely (from 6 to over 500).

1.5 Discussion:

Pereira ([Pereira, 1985]) attains structure-sharing by having the result graph share information with the original graphs by storing changes to the 'environment'. There will be the log(d) overhead (where d is the number of nodes in a graph) associated with Pereira's method that is required during node access 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, 1986]'s reversible unification and copy graphs only after a successful unification. However, Karttunen's method does not use structure-sharing. Also, In Karttunen's method[31], whenever a destructive change is about to be made, the attribute value pairs[32] stored in the body of the node are saved into an array. The dag node structure itself is also saved in another 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 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 the changes made to the nodes. [Karttunen and Kay, 1985] suggests the use of lazy evaluation to delay destructive changes during unification. [Godden, 1990] presents 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

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

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

of delayed evaluation functionalities in programming languages, actual efficiency gain from such a scheme may not be significant. This is because such a scheme simply shifts the time and space consumed for copying to creating and evaluating closures (which could be very costly compared to 'defstruct' operations to create copies which are often effectively optimized in many commercial compilers). [Kogure, 1990] and [Emele, 1991] also use the lazy evaluation idea to delay destructive changes. Both Kogure and Emele avoid direct usage of delayed evaluation by using pointer operations. As Emele suggests, Kogure's method also requires a special dependency information to be maintained which adds an overhead along with the cost for 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) to find the node that shares the same generation number as the root node. Therefore, similar to Pereira's method, there will be Nlog(d) overhead associated with constructing each graph every time a graph is accessed, where d is the number of nodes in the graph and N is the average depth of the environmental deference chain. This would cause a problem if the algorithm is adopted for a large-scale system in which result graphs are unified against other graphs many times. 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 [Tomabechi, 1991]. This is because the copies that are incrementally created up to the point of failure during the same top-level unification are wasted. The problem is inherent in incremental copying scheme and this problem is eliminated completely in [Karttunen, 1986] and in the Q-D method.[33]

[33] Lazy methods delay copying until a destructive change is to be performed so that unnecessary copies are not created within a particular recursion into a unification function; however, since each shared arc recursion is independent (non-deterministic), even if there are no unnecessary copies created at all in one particular recursion, if there is a failure in some other shared arc recursion (at some depth), then the copies that are created by successful shared arc recursions up to the point of detection of failure will become wasted. As long as the basic control structure remains incremental, this is inherent in the incremental method. In other words, the problem is inherent in these incremental

There is one potential problem with the structure-sharing idea which is shared by each of the schemes including the proposed method. This happens when operations other than unification modify the graphs. (This is typical when a parser cuts off a part of a graph for subsequent analysis[34].) When such operations are performed, structure-sharing of bottom (variable) nodes may cause problems when a subgraph containing a bottom is shared by two different graphs and these graphs are used as arguments of a unification function (either as the part of the same input graph or as elements of dg1 and dg2). When a graph that shares a bottom node is not used in its entirety, then the represented constraint postulated by the path leading to the bottom 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 bottom node, there will be a false 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 bottom node was unified against some other graph before appearing in the same unification.[35] (If either was once unified, forwarding would have avoided this problem). The methods to avoid such a problem can be 1) As long as these convergence of bottom nodes are used for features that are not passed up during parsing, the problems does not affect the result of parse in any way – which is the case with the grammar at ATR and CMU. 2) A parser can be modified so that when it modifies a graph other than through graph unification[36], it creates copies of the arc structures containing the bottom nodes. In the proposed method this can be done by calling the copy function without structure-sharing before a parser modifies a graph. 3) A parser can be modified so that it does not cut off parts of graphs and use the graphs in their entirety (this should not add complexity once structure-sharing is introduced to unification). Thus, although the space and time reduction attained by structure-sharing can be significant, DS-Sharing can cause problems unless it is used with a caution (by making sure variable sharing does not cause erroneous sharing by using these or some other methods).

---

methods by definition.

[34]For example, many parsers cut off a subgraph of the path 0 for applying further rules when a rule is accepted.

[35]Such cases may happen when the same rule (such as V $\Rightarrow$ V) augmented with a heavy use of convergence in the bottom nodes is applied many times during a parse.

[36]Such as when a rule is accepted and subgraph of 0 path is cut off.

1.6 Conclusion

The structure-sharing scheme introduced in this paper made the Q-D algorithm (which was already significantly faster than Wroblewski's non-destructive unification) run significantly faster. 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 incremental copying scheme inherently produces Early Copies (as defined in [Tomabechi, 1991]) when a unification fails. The proposed scheme makes the Q-D algorithm fully avoid Redundant Copies as well by only copying the lowest nodes that need to be copied due to destructive changes caused by successful unifications only. Since there will be no overhead associated with structure-sharing (except for returning two values instead of one to pass up :changed information when recursion for copying returns), the performance of the proposed structure-sharing scheme should not drop even when the grammar size is significantly scaled up.[37] With the demonstrated speed of the algorithm, as well as the ability to handle cyclicity in the graphs, and ease of switching between structure-sharing and non-structure sharing, the algorithm could be a viable alternative to existing unification algorithms used in current natural language systems.

ACKNOWLEDGMENTS

The author would like to thank Akira Kurematsu, Tsuyoshi Morimoto, Hitoshi Iida, Osamu Furuse, Masaaki Nagata, Toshiyuki Takezawa and other members of ATR and Masaru Tomita, Jaime Carbonell, Alex Waibel and David Evans at CMU. Kiyoshi Kogure of NTT (formally with ATR) implemented the original versions of the Earley's algorithm and Wroblewski's algorithm used in the experiments reported in this paper. The author also had a number of useful discussions with Kogure concerning structure-sharing and graph unification algorithms. A number of researchers and programmers at CMU have contributed in implementing the original Tomita generalized LR parser which now integrates the Q-D algorithm with structure-sharing. Thanks are also due to Margalit Zabludowski for comments on the final version of this paper and Madoka Higuchi for assistance in preparing the final version of the paper.

[37] ATR has already chosen the proposed structure-sharing scheme to be integrated into their implementation of the Q-D algorithm already adopted for their large scale speech-to-speech translation project.

Implementation

The unification algorithms, Earley parser and the HPSG path equation to graph converter programs are implemented in CommonLisp on a Symbolics machine. A grammar compiler has also been implemented for the Tomita generalized LR parser using CommonLisp to integrate Q-D unification with structure-sharing into CMU's Universal Parser (v8.4) architecture ([Tomita and Carbonell, 1987]).

Bibliography

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

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

[Karttunen, 1986] 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.

[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.

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

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

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

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

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

Chapter 2

破壊型単一化から準破壊型単一化へ

2.1 自然言語処理におけるグラフの単一化の役割

- "In the past several years, theoretical and computational linguists working within a number of distinct research traditions have found it useful to construct formal models of linguistic information" – Pollard.

- 素性構造の単一化を形式化のベースとした言語理論、計算言語理論 – FUG (Kay, 1984), LFG (Bresnan and Kaplan, 1982), GPSG (Gazdar, etal, 1985), HPSG (Pollard and Sag, 1987).

- 制約記述に有向グラフを利用した超並列自然言語処理 – MONA-LISA (Tomabechi, 1990, 1991), GCPN (Tomabechi, 1991).

2.2 グラフ単一化の問題点 – 計算量（実行速度）

- グラフ単一化は一般にパーザーの処理実時間の 85% から 95% ぐらいの時間を消費している。

- 従って、単一化アルゴリズムの高速化はパーザーの高速化に最もインパクトを与えるはずである。

2.3 高速なグラフ単一化手法に向けた課題

- 入力グラフを破壊してはならない。
- 出力グラフが将来の処理で利用される。
- グラフ中にループがあることがある。
- 端末のノードが変数であることがある。
- 効率的なデータ構造とその使用（ガベコレしやすい等を含む）。

2.4 **Pereira** の手法

- Structure-sharing の考え方。
- オリジナルのグラフと結果のグラフが変更部以外を共有する。
- 従って、結果のグラフはオリジナルのグラフと変更情報の結合として表される。
- *Skelton* と *Environment* のデータ構造。
- グラフのコピーを作らず必要に応じてグラフをダイナミックに作製する。
- 従って、log(d) の固定的なオーバーヘッドがノードへのアクセス毎に必要である。ここで、 d はグラフ中のノードの数。

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

Figure 2.1: **Pereira's Data Structure**

2.5 Kartunnen の手法

- 再生型単一化 (reversible unification) の考え方。
- 破壊的な変更を行なう直前にアレイに内容をセーブ。
- D-PATR (C-PATR): 二つのアレイ。ひとつにノード構造をセーブ、もうひとつに構造の内容をセーブ。
- 単一化の成功後にコピーを作る。
- 一回の単一化終了毎にオリジナルのグラフを再生する。
- 実際にコピーを作製するので、ノードへのアクセス毎のオーバーヘッドはない。
- 一方、破壊的変更時のグラフのセーブと、単一化後のグラフの再生にオーバーヘッドがかかる。

2.6 Wroblewski の手法

- 進行型複製 (incremental copying—) の考え方。
- 非破壊型の単一化 (破壊型単一化と非破壊型単一化の混在)
- set-difference の集合演算を dag1 と dag2 の両方向から行なう。
- ノードアクセスへのオーバーヘッドはない。
- 現在広く利用されている。 - Godden, Kogure, Emele, etc.

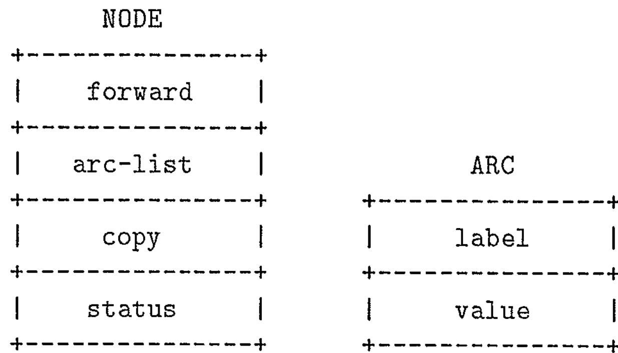


Figure 2.2: **Wroblewski's Data Structure**

2.7 高速な汎用グラフ単一化アルゴリズムの提案

- 狙い
  - Wroblewski のアルゴリズムの 2 倍以上の実処理速度を狙う。
  - DAG ではなく DG を扱う。 ⟹ 一般的な有向グラフを扱う。
  - 並列化しやすいアルゴリズム。

2.8 高速化問題の切口

- 単一化は失敗することがある。
- 複製の作製は重たい。
- 集合演算は重たい。（特に Incremental Copying 方式では、両方向の set-difference をとる必要がある。）
- 無変更のグラフは共有できる。

2.9 単一化の失敗

- 60 %以上の単一化が失敗していた。
- 単一化が失敗した場合その間の操作は無駄になる。

2.10 複製行為の負荷

- 時間と空間を消費する。 – メモリーのアロケーション、ページング等。
- ガーベージコレクションがより頻繁になる。
- 単一化に要する時間の７５％から９５％が複製の作製につかわれている。

2.11 無変更のグラフの共有化

- 有向グラフは一方向への参照である。
- 破壊的変更時には複製が作製される。
- 未変更のサブグラフは共有できるはず。

2.12 解決への方向

- コピーは単一化成功時のみに作製する。
- 単一化失敗はできる限り早期発見。
- オーバーヘッドのない構造共有を目指す。

2.13 Over-Copying と Early-Copying

- Over Copying: 一つの結果ノードを作製するために、二つの複製ノードを作製してしまう。 – 破壊的単一化の直前に両入力グラフを複製するような手法で見られる。
- Early Copying: 単一化の失敗発見以前に複製が作製される。単一化の開始から失敗の発見までに無駄な複製がつくられる。 – Wroblewski の定義と違うことに注意。

Wroblewski の Early Copying の定義: "The argument dags are copied *before* unification started. If the unification fails then some of the copying is wasted effort". このように Wroblewski の定義では単一化が開始してから失敗の発見までに作製されたコピーが Over Copying にも Early Copying にも属さなくなってしまう。

2.14 進行型複製手法 (Incremental Copying) の本質的問題

- 前項での定義による Early Copying が解決不能。

  Shared Arc への再帰は non-deterministic である。したがって、ある Arc への単一化の再帰のひとつが失敗した場合他の成功した再帰中にインクリメンタルに作製された複製は全て無駄になる。

- 同様に再帰が non-deterministic であるから、 Convergence や cycle をグラフに認めた場合、部分的構造の共有化をするためには、各再帰中で作製された構造共有部を独立した再帰間で共有するため、ポインターなどを利用した再帰間のコミュニケーションが必要となる。

- これらは、 Wroblewski, Kogure, Godden, Emele 等全ての進行型複製手法を採るアルゴリズムに本質的に (Incremental の定義上) 共通である。

- このように、進行型複製手法では 1) Early Copying が解決されない、 2) 構造共有のオーバーヘッドが大きい 3) 並列化が難しいという問題がある。

2.15 基本的手法

- 時間差的内容表現 - グラフの表現内容がグローバルクロックに依存する。

- 破壊的変更を安く (constant time) 時間差的に打ち消す。

- アーク、フォワードポインター、複製ポインター等の表現内容が全て時間に依存。

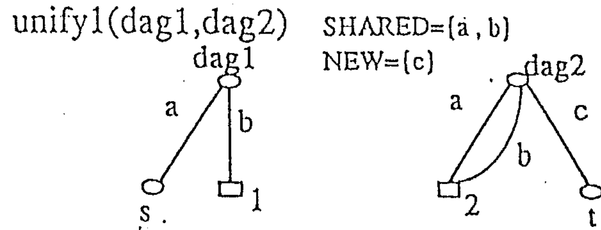
- トップレベルの単一化関数呼びだしが時間の単位。

- 一つの時間単位中に行なわれた全ての破壊的変更はグローバルクロックが一単位進めばキャンセルされる。

2.16 ループあるいはリエントランシーと一時的フォワード

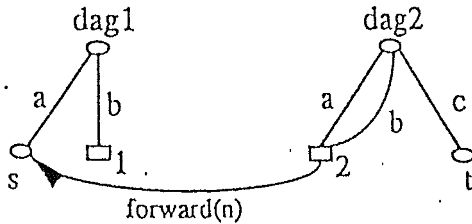
- unify1 が成功した場合、 dg2 から dg1 へ一時的フォワードのリンクが張られる。

2.17 時間差準破壊型手法による単一化例



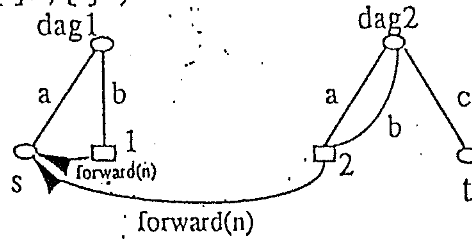
For each node with arc-a,

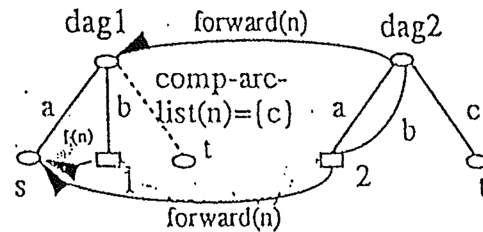
unify1(s, [ ]2)



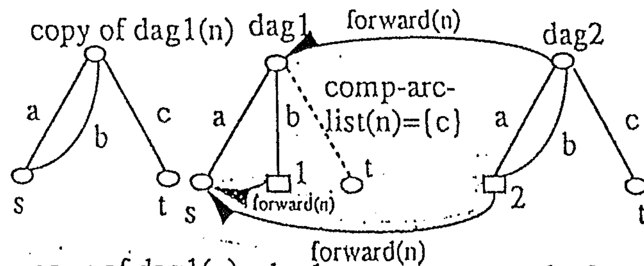
For each node with arc-b,

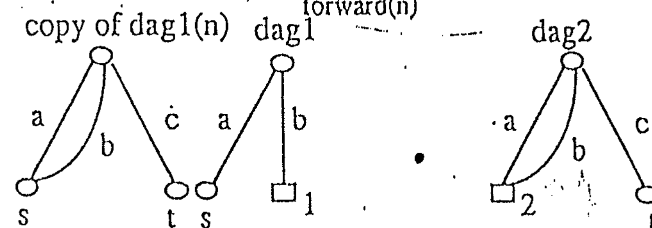
unify1([ ]1, [ ]2)





copy-comp-arc-list(dag1)





2.17 時間差準破壊型手法による単一化例

2.18 時間差準破壊型アルゴリズムの単一化における特徴

- 非破壊的である。

- Set-difference の集合演算は一方向からだけ, また、サブグラフが全て成功した時のみ行なわれる。

  “こちらからそちらへ登録用紙を至急お送り致します”の解析実験で、Q-D 手法では、unify1 が３１１３回呼ばれ、intersectarcs が１７４７回、complementarcs が６６０回呼ばれている。Incremental Copying の手法では、intersectarcs が１７４７回、complementarcs が３４９４回呼ばれたはずである。したがって、２８３４回の集合演算が省かれたことになる。

- 複製は成功した時のみ作られるので、Over Copying、Early Copying ともに解消される。

- オリジナルグラフの再生にオーバーヘッドがない。時計を一単位進めるだけ。

- ループやリエントランシーが特別な処理なしに自然に扱える。

2.19 グラフ単一化におけるデータ構造の共有： 定義

- **Feature-Structure Sharing:** – 文法中のリエントランシーで表されるような、素性構造の内容上の構造共有。同一のグラフの中で二つ以上の経路が同じノードへ行っている場合。

- **Data-Structure Sharing:** – Pereira のアルゴリズムに見られるような、データ構造上の Structure-Sharing。二つ以上の異なるグラフが同じノードを共有している場合。

2.20 小暮の手法 (Coling'90)

- Copy-dependency と呼ばれるスロットに (arc と mother) のペアのリストを格納しておく。

- 基本的には Wroblewski のアルゴリズムと同様だが、Complementarcs のノードを即時複製せずに、Copy-dependency に作製されるべきノードを書き込んでいきそれぞれの局所的な再帰が戻る時点まで複製を遅らせる。再帰が戻る時に変更を受けるべきノードのみ複製する。

- 利点：

  – ノードアクセスのオーバーヘッドなしに無変更のサブグラフを共有可能。

  – Copy-dependency 情報をノードに直接書き込むことにより、ループやリエントランシーがあった場合も、対処可能である。

- 問題点：

  – Early Copy は完全になくならない。– それぞれの局所的再帰は non-deterministic であるので、複製を局所的に遅らせても、失敗が発見されなかった全ての局所的な再帰でつくられた複製が無駄になる。

  – Copy-dependency 情報を維持しなければならないため、その参照、解読のオーバーヘッドが対数的にでる。また、早期 GC が難しい。

2.21 Emele の手法 (ACL'91)

- 時系列フォワードの手法 (chronological dereferencing) 。
- Wroblewski と同様な Incremental Copying. 小暮アルゴリズムのような、遅延は行なわない。
- ノードの複製のみつくりアークは複製しない。
- 必要なグラフは各ノードをルートが持つ時系列の時点の ID と同じ ID が得られまで、鎖状に dereference することで作製される。
- 利点：

  – 変更のあったノードまでいたる経路状のノードまでも共有される。

  – アークは一切複製されない。

- 問題点：

  – Incremental Copying であるため、Early Copying は解決されない。

  – グラフのアクセス毎に dereference chain を解読しグラフを作らねばならないため、Pereira のアルゴリズムと同様 log(d) のオーバーヘッドがかかり、それぞれのノードにおけるオーバーヘッドも dereference chain の深さに比例してかかる。– Nlog(d)。

– 大規模な文法では、同じグラフが多数回利用されまた、作製されたグラフが多数回異なるグラフと単一化される。従って、 chronological dereference に要するオーバーヘッドは極めて大きくなると予想され、実用には向かないと思われる。

2.22 時間差準破壊型手法における構造の共有 – 基本的な考え方

- 無変更のノードの共有
- 無変更の定義：

– 全ての Atomic ノード （Atomic な値は不変）。

– 全ての Leaf ノード （単一化以外にグラフを変更することがないことが前提）。

– 全ての子孫が無変更の Complex ノード。

– 但し、ノードがフォワードの行き先であったときは変更があったとみなす。

2.23 時間差準破壊型手法における構造の共有 – アルゴリズムの変更

- 単一化関数（unify0, unify1 等） – 変更不要。
- 複製関数 – 以下の変更

– 複製関数が再帰から戻る時に、複製を値として返すのみでなく、そのノードもしくはそのノードの子孫が変更を受けたかのフラグを返す。

– もしそのノード自身と子孫の全てが無変更であり、かつそのノードがフォワードの行き先でなければ、複製を作らずオリジナルを:unchanged のフラグと共に返す。変更があれば複製を作り:changed のフラグと共に返す。

– もしそのノード自身と子孫の全てが無変更であり、かつそのノードがフォワードの行き先であれば、複製を作らずオリジナルを:changed のフラグと共に返す。変更があれば複製を作り:changed のフラグと共に返す。

2.24 構造共有化 (Data-Structure Sharing) の一つの問題点

- 自然言語システムにおいて単一化だけがグラフを変更するとは限らない。
- 例えば、一般にパーザーは CFG ルールが成功すると、X0 のパスを切り取る。
- この場合 Bottom の DS-Sharing により誤った FS-Sharing が発生することがあり得る。

2.25 Bottom 共有による問題の解決法：(いくつかの手法例)

- 単一化アルゴリズム中において特定のラベル付のアーク (例えば X0) をコピーする場合に DS-Share しないコピーを行なう。 – Complex ノードもそのまま複製する (Complex ノードの中に bottom があるかもしれないから)
- Bottom の DS-Sharing はしない。この手法と前項の手法のそれぞれのトレードオフはどの程度 Complex ノードの Sharing がきいているかできまる。例えば、文法全体で同一のテンプレートで表されるようなグラフを DS-Share しているとすれば、Bottom の DS-Sharing をやめ、XO グラフの切り取り時も構造共有すべき。
- パーザーがグラフを切り取るのをやめる。単一化は monotonic な操作であり、Bottom の DS-Sharing が問題を起こすのは情報の単調増加の原則を崩すためである。つまり単一化の結果のグラフのもつ情報が入力のグラフの情報より少ないという現象を起こしているからである。
- 文法によっては、不要な Bottom の DS-Sharing ができてもヘッドの情報として上がっていかないこともがあり、こういった文法では全く結果には影響を及ばさない。

2.26 結論

- 単一化の再帰時に複製作製等のオーバーヘッドがなく失敗の早期が可能。
- Early Copying、Over Copying 共に解消。

- アルゴリズム的追加なく自然にループやリエントランシーが扱える。
- アクセス時も複製作製時もオーバーヘッドのない構造共有 (DS-Sharing)
- 単一化関数群の変更なしに複製関数をスイッチするだけで、構造の共有と非共有を混在可能 ⇐ 従って、パーザがグラフを破壊的に変化させる場合にも対応できる。
- 実験で構造非共有時で Wroblewski の 2 倍から 3 倍、構造共有時で 4 倍から 5 倍程度の実処理速度であることが観察された。
- また、単一化の失敗の割合が上がれば上がるほど Early Copying が本質的に解消できない Incremental Copying を利用した手法との差が大きくなるため、一般に文法のスケールが上がれば上がるほど、高速化の効果があがるものと見られる。
- 基本的には縦横両方向に並列化可能であるはずであるが、残念ながら昨年度以来、並列アルゴリズムの実験は行なっていない。ただし、構造共有時には、同じ構造への複数プロセスのアクセスによるロックの問題が顕著になり得るため、トレードオフがあるはず。
- グラマーコンパイラーを書き換えることによって、文法全体を通してののATOM やテンプレートで書かれるような COMPLEX のグラフの共有が可能になる。