In proceedings of the Inter. Workshop on Parsing technologies '91 164-176 (IWPT'91)

Quasi-Destructive Graph Unification*

Towards Efficiency and Parallelism in Graph Unification

Hideto Tomabechi

Carnegie Mellon University
109 EDSH, Pittsburgh, PA 15213-3890, U. S. A.
tomabech@cs.cmu.edu

Abstract

Provided graph unification shares majority of parsing time in a typical natural language system it makes sense to improve efficiency of parsing algorithms to improve the overall efficiency of the parsing systems. In this paper, we focus on two speed-up elements in the design of unification algorithms: 1) elimination of excessive copying by only copying successful unifications, 2) Finding unification failures as soon as possible. We have developed a scheme to attain these two criteria without expensive overhead through temporarily modifying graphs during unification to eliminate copying during unification. The temporary modification is invalidated in constant time and therefore, unification can continue looking for a failure without the overhead associated with copying. After a successful unification because the nodes are temporarily prepared for copying, a fast copying can be performed without overhead for handling reentrancy, loops and variables. Our experiments show a significant speed up over Wroblewski's algorithm in an overall parsing time by the adoption of the proposed methodology.

*The author would like to thank Akira Kurematsu, Tsuyoshi Morimoto, Hitoshi Iida, Osamu Furuse, Masaaki Nagata, Toshiyuki Takezawa and other members of ATR. Thanks are also due to Margalit Zabludowski of Carnegie Mellon University for comments on the final version of this paper and Takako Fujioka of Waseda University for assistance in implementing the parallel version of our algorithm.

1. Motivation

One area of parsing technology is a method of graph unification. It is indeed the most expensive part of the overall natural language processing. For example, in the three types of unification-based parsing systems currently used at ATR[1], all of which use graph unification algorithms based on [Wroblewski, 1987], unification operations consume 85 to 90 percent of the total cpu time devoted to a parse. The number of unification operations per sentence tends to grow as the grammar gets larger and more complicated. An unavoidable paradox is that when the natural language system gets larger and the coverage of linguistic phenomena increases the writers of natural language grammars tend to rely more on deeper and more complex path equations (loops and frequent reentrancy) to lessen the complexity of writing the grammar. As a result, we have seen that the number of unification operations increases rapidly as the coverage of the grammar grows in contrast to the parsing algorithm itself which does not seem to grow so quickly. Thus, it makes sense to speed up the unification operations to improve the total speed performance of the natural language parsing system.

Our original unification algorithm was based on [Wroblewski, 1987] which was chosen in 1988 as the then fastest algorithm available for our application (HPSG based unification grammar, three types of parsers (Earley, Tomita-LR, and active chart), unification with variables and loops[2] combined with Kasper's ([Kasper, 1987]) scheme for handling disjunctions). In designing the graph unification algorithm, we have made the following observation which influenced the basic design of the new algorithm described in this paper:

Unification does not always succeed.

As we will see from the data presented in a later section, when our parsing system operates with a relatively small grammar, about 60 percent of unifications attempted during a successful parse result in failure. If a unification fails, any computation performed and memory consumed during the unification is wasted. As the grammar size increases, the number of unification failures for each successful parse increases[3]. Without completely rewriting the grammar and the parser, it seems difficult to shift any significant amount of the computational burden to the parser in order to reduce the number of unification failures[4].

Another problem that we would like to address in our design, which seems to be well documented in the existing literature is that:

Copying is an expensive operation.

The copying of a node is a heavy burden to 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 essentially because the area in the random access memory needs to be dynamically allocated which is an expensive operation. [Godden, 1990] calculates the computation time cost of copying to be about 67 % of total parsing time in his TIME parsing system. This time/space burden of copying is non-trivial when we consider the fact that creation of unnecessary copies will eventually trigger garbage collections more often (in a Lisp environment) which will also slow down

[1] The three parsing systems are based on: 1. Earley's algorithm, 2. active chart parsing, 3. generalized LR parsing.

[2] Please refer to [Kogure, 1989] for trivial time modification of Wroblewski's algorithm to handle loops.

[3] We estimate over 80% of unifications to be failures in our large-scale speech-to-speech translation system under development.

[4] Of course, whether that will improve the overall performance is another question.

the overall performance of the parsing system. In general, parsing systems are always short of memory space (such as large LR tables of Tomita-LR parsers and expanding tables and charts of Earley and active chart parsers[5]), and the marginal addition or subtraction of the amount of memory space consumed by other parts of the system often has critical effects on the performance of these systems.

Considering the aforementioned problems, we propose the following principles to be the desirable conditions for a fast 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 dags into 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 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. ([Godden, 1990] calls such a unification: Eager Unification.). When two arcs point to the same node, over copying is often unavoidable with incremental copying schemes.
- **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" and restricts early copying to cases that only apply to copies that are created prior to a unification. Restricting early copying to copies that are made prior to a unification leaves a number of wasted copies that are created during a unification up to the point of failure to be uncovered by either of the above 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 actual unification operations.

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 cases of convergent arcs into one node), incremental copying is ineffective in eliminating early copying as we define it.[6] Incremental copying is ineffective in eliminating early copying because when a graph unification algorithm recurses for shared arcs (i.e. the arcs with labels that exist in both input graphs), each created unification operation recursing into each shared arc is independent of other recursive calls into other arcs. In other words, the recursive calls into shared arcs are non-deterministic and there is no way for one particular recursion into a shared arc to know the result of future recursions into other shared arcs. Thus 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 and thus the copies that are created in the successful arcs are all wasted. We consider it a drawback of incremental copying schemes that copies that are incrementally created up to the point of failure get wasted. This problem will

[5]For example, our phoneme-based generalized LR parser for speech input is always running on a swapping space because the LR table is too big.

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

be particularly felt when we consider parallel implementations of incremental copying algorithms. Because each recursion into shared arcs is non-deterministic, parallel processes can be created to work concurrently on all arcs. In each of the parallelly created processes for each shared arc, another recursion may take place creating more parallel processes. While some parallel recursive call into some arc may take time (due to a large number of subarcs, etc.) another non-deterministic call to other arcs may proceed deeper and deeper creating a large number of parallel processes. In the meantime, copies are incrementally created at different depths of subgraphs as long as the subgraphs of each of them are unified successfully. This way, when a failure is finally detected at some deep location in some subgraph, other numerous processes may have created a large number of copies that are wasted. Thus, early copying is a devastaring problem when we consider parallelization of incremental copying unification algorithms.

2. Proposed Scheme

In this section, we would like to propose an algorithm meeting the criteria for fast unification discussed in the previous sections. It also handles loops without over copying (without any additional schemes such as those introduced by [Kogure, 1989]).

As a data structure, a node is represented with eight fields: type, arc-list, comp-arc-list, forward, copy, comp-arc-mark, forward-mark, and copy-mark. Although this number may seem high for a graph node data structure, the amount of memory consumed is not significantly different from that consumed by other algorithms. Type can be represented by three bits; comp-arc-mark, forward-mark, and copy-mark can be represented by short integers (i.e. fixnums); and comp-arc-list (just like arc-list) is a mere collection of pointers to memory locations. Thus this additional information is trivial in terms of memory cells consumed and because of this data structure the unification algorithm itself can remain simple.

```
      NODE
+---------------+
|     type      |
+---------------+
|   arc-list    |
+---------------+
| comp-arc-list |
+---------------+
|    forward    |
+---------------+
|     copy      |
+---------------+
| comp-arc-mark |
+---------------+
| forward-mark  |
+---------------+
|   copy-mark   |
+---------------+
```

Figure 1: **Node Structure**

The representation for an arc is no different from that of other unification algorithms. Each arc has two fields for 'label' and 'value'. 'Label' is an atomic symbol which labels the arc, and 'value' is a pointer to a node. The central notion of our algorithm is the dependency of the representational

```
        ARC
+---------------+
|     label     |
+---------------+
|     value     |
+---------------+
```

Figure 2: **Arc Structure**

content on the global timing clock (or the global counter for the current generation of unification algorithms). This scheme was used in [Wroblewski, 1987] to invalidate the copy field of a node after one unification by incrementing a global counter. This is an extremely cheap operation but has the power to invalidate the copy fields of all nodes in the system simultaneously. In our algorithm, this dependency of the content of fields on global timing is adopted for arc lists, forwarding pointers, and copy pointers. Thus any modification made, such as adding forwarding links, copy links or arcs during one top-level unification (unify0) to any node in memory can be invalidated by one increment operation on the global timing counter. During unification (in unify1) and copying after a successful unification, the global timing ID for a specific field can be checked by comparing the content of mark fields with the global counter value and if they match then the content is respected, if not it is simply ignored. Thus the whole operation is a trivial addition to the original destructive unification algorithm (Pereira's and Wroblewski's unify1).

We have two kinds of arc lists 1) arc-list and comp-arc-list. Arc-list contains the arcs that are permanent (i.e., usual graph arcs) and comp-arc-list contains arcs that are only valid during one graph unification operation. We also have 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 unification. The currency of the temporary links is determined by matching the content of the mark field for the links with the global counter and if they match then the content of this field is respected[7]. As in [Pereira, 1985], we have three types of nodes: 1) :atomic, 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. Arc values are always nodes and never symbolic values because the :atomic and :bottom nodes may be pointed to by multiple arcs (just as in structure sharing of :complex nodes) depending on grammar constraints, and we do not want arcs to contain terminal atomic values.

Below is our algorithm:

```
;;;UNIFY-DAG is the top level entry. It calls UNIFY0 which calls UNIFY1.
```

[7] In terms of forwarding links, 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 forward-mark is not 9 the integer value must equal the global counter value to respect the forwarding link.

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

```
 ;;;If UNIFY1 succeeds, a copy is returned to UNIFY-DAG. If UNIFY1 fails at
 ;;;any time of recursion, it immediately returns 'UNIFY-FAIL to UNIFY-DAG.
function UNIFY-DAG (dag1,dag2);
 RESULT :== catch with tag 'UNIFY-FAIL calling UNIFY0(dag1,dag2)
 increment *unify-global-counter*  ;;; initially, this value starts from 10
 return RESULT;
end;

function UNIFY0 (dag1,dag2);
  if '*T* ==  UNIFY1(dag1,dag2);
      then COPY :== COPY-DAG-WITH-COMP-ARCS(dag1);
           return COPY;
end;

function UNIFY1 (dag1-underef,dag2-underef);
 DAG1 :== DEREFERENCE-DAG(dag1-underef);
 DAG2 :== DEREFERENCE-DAG(dag2-underef);

 if (DAG1 == DAG2)            ;;; i.e., 'eq' relation, this may happen
    then return '*T*;         ;;; because of forwarding and loops

 else if (DAG1.type == :bottom) ;; i.e., variable
        then FORWARD-DAG(DAG1,DAG2,:temporary);
             return '*T*;

 else if (DAG2.type == :bottom)
        then FORWARD-DAG(DAG2,DAG1,:temporary);
             return '*T*;

 else if (DAG1.type == :atomic and DAG2.type == :atomic)
        then
            if (DAG1.arc-list == DAG2.arc-list)  ;;;contains atomic values
               then FORWARD-DAG(DAG2,DAG1,:temporary);
            return '*T*;
            else throw with keyword 'UNIFY-FAIL;
                ;;; i.e., return directly to unify-dag (throw/catch construct)

 else if (DAG1.type == :atomic or DAG2.type == :atomic)
        then
            throw with keyword 'UNIFY-FAIL;

 else  NEW :== COMPLEMENTARCS(DAG2,DAG1);
       SHARED :== INTERSECTARCS(DAG1,DAG2);
       for each ARC in SHARED do
          RESULT :== UNIFY1(destination of the shared arc for dag1,
                            destination of the shared arc for dag2);
          if (RESULT == '*T*)
             then
                return '*T*;
             else throw with keyword 'UNIFY-FAIL;

       If (the recursive calls to UNIFY1 successfully returned
           for all shared arcs) ;;; this check is actually unnecesary because if there is
          then                  ;;; a failure, unify1 has returned already (by throw).
             FORWARD-DAG(DAG2,DAG1,:temporary);
             DAG1.comp-arc-mark :==  *unify-global-counter*;
             DAG1.comp-arc-list :== NEW
             return '*T*;
end;
```

```
function COPY-DAG-WITH-COMP-ARCS (dag-underef);
 DAG :== DEREFERENCE-DAG(dag-underef);
 if (DAG.copy is non-empty
       and
     DAG.copy-mark == *unify-global-counter*)
    then return the content of DAG.copy;   ;;; i.e. existing copy

 else if (DAG.type == :atomic)
        COPY :== CREATE-NODE();      ;;; creates an empty node.
        COPY.type :== :atomic;
        COPY.arc-list :== DAG.arc-list;    ;;; this is an atomic value
        DAG.copy :== COPY;
        DAG.copy-mark :== *unify-global-counter*;
        return COPY;

 else if (DAG.type == :bottom)
        COPY :== CREATE-NODE();
        COPY.type :== :bottom;
        DAG.copy :== COPY;
        DAG.copy-mark :== *unify-global-counter*;
        return COPY;

 else
       COPY :== CREATENODE();
       COPY.type :== :complex;
       for all ARC in DAG.arc-list do
             NEWARC :== COPY-ARC-AND-COMP-ARC(ARC);
             push NEWARC into COPY.arc-list;
       if (DAG.comp-arc-list is non-empty
                  and
           DAG.comp-arc-mark == *unify-global-counter*)
          then
             for all COMP-ARC in DAG.comp-arc-list do
                 NEWARC :== COPY-ARC-AND-COMP-ARC(COMP-ARC);
                 push NEWARC into COPY.arc-list;

       DAG.copy :== COPY
       DAG.copy-mark :== *unify-global-counter*;
       return COPY;
  end;

function COPY-ARC-AND-COMP-ARC (input-arc)
 LABEL :== label of input-arc;
 VALUE :== COPY-DAG-WITH-COMP-ARCS(value of input-arc);
 return a new arc with LABEL and VALUE;
 end;
```

The functions Complementarcs(dag1,dag2) and Intersectarcs(dag1,dag2) are the same as in Wroblewski's algorithm and return the set-difference (the arcs with labels that exist in dag1 but not in dag2) and intersection (the arcs with labels that exist both in dag1 and dag2) respectively. Dereference-dag(dag) recursively traverses the forwarding link to return the forwarded node. In doing so, it checks the forward-mark of the node and if the forward-mark value is 9 (9 represents a permanent forwarding link) or its value matches the current value of *unify-global-counter*, then the function returns the forwarded node; otherwise it simply returns the input node. Forward(dag1, dag2, :forward-type) puts (the pointer to) dag2 in the forward field of dag1. If the keyword in the function call is :temporary, the current value of the *unify-global-counter* is written in the forward-mark field of dag1. If the keyword is :permanent, 9 is written in the forward-mark field of dag1. Our algorithm itself does not require any permanent forwarding; however, the functionality is added because the grammar reader module that reads the path equation specifications into dag feature-structures uses permanent forwarding to merge the additional grammatical specifications into a graph structure[9]. The temporary forwarding links are necessary to handle reentrancy and loops. As soon as unification (at any level of recursion through shared arcs) succeeds, a temporary forwarding link is made from dag2 to dag1 (dag1 to dag2 if dag1 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 dag2 to dag1 (or dag1 to dag2). As a result, if this node becomes an input argument node, dereferencing the node causes dag1 and dag2 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 loop. Also, during copying done subsequently to a successful unification, two arcs converging into the same node will not cause over copying simply because if a node already has a copy then the copy is returned. For example, as a case that may cause over copies in other schemes for dag2 convergent arcs, let us consider the case when the destination node has a corresponding node in dag1 and only one of the convergent arcs has a corresponding arc in dag1. This destination node is already temporarily forwarded to the node in dag1 (since the unification check was successful prior to copying). Once a copy is created for the corresponding dag1 node and recorded in the copy field of dag1, every time a convergent arc in dag2 that needs to be copied points to its destination node, dereferencing the node returns the corresponding node in dag1 and since a copy of it already exists, this copy is returned. Thus no duplicate copy is created[10].

As we just saw, the algorithm itself is simple. The basic control structure of the unification is similar to Pereira's and Wroblewski's unify1. The essential difference between our unify1 and the previous ones is that our unify1 is non-destructive. It is because the complementarcs(dag2,dag1) are added to the comp-arc-list of dag1 and not into the arc-list of dag1. Thus, as soon as we increment the global counter, the changes made to dag1 (i.e., addition of complement arcs into comp-arc-list) vanish. As long as the comp-arc-mark value matches that of the global counter the content of the comp-arc-list can be considered a part of arc-list and therefore, dag1 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 dag1 before we increment the global counter while respecting the content of the comp-arc-list of dag1.

Thus instead of calling other unification functions (such as unify2 of Wroblewski) for incrementally creating a copy node, during a unification, we only need to create a copy after unification. Thus, if unification fails no copies are made at all (as in [Karttunen, 1986]'s scheme). Because unification that recurses into shared arcs carries no burden of incremental copying (i.e., it simply checks if nodes are compatible), as the depth of unification increases (i.e., the graph gets larger) the speed-up of our method should get conspicuous if a unification eventually fails. If all unifications during a parse are going to be successful, our algorithm should be as fast as or slightly slower than

---

[9] We have been using Wroblewski's algorithm for the unification part of the parser and thus usage of (permanent) forwarding links is used by the grammar reader module.

[10] Copying of dag2 arcs happens for arcs that exist in dag2 but not in dag1 (i.e., Complementarcs(dag2,dag1)). Such arcs are pushed to the comp-arc-list of dag1 during unify1 and are copied into the arc-list of the copy during subsequent copying. If there is a loop or a convergence in arcs in dag1 or in arcs in dag2 that do not have corresponding arcs in dag1, then the mechanism is even simpler than the one discussed here. A copy is made once, and the same copy is simply returned every time another convergent arc points to the original node. It is because arcs are copied only from either dag1 or dag2.

Wroblewski's algorithm[11]. Since a parse that does not fail on a single unification is unrealistic, the gain from our scheme should depend on the amount of unification failures that occur during a unification. As the number of failures per parse increases and the graphs that failed get larger, the speed-up from our algorithm should become more apparent. Therefore, the characteristics of our algorithm seem desirable.

3. Experiments

sent#	Unifs	USrate	Elapsed time(sec)		Num of Copies	
			T	W	T	W
1	6	0.5	1.270	1.215	85	107
2	101	0.35	2.143	2.806	1418	2285
3	24	0.33	1.325	1.312	129	220
4	71	0.41	3.673	4.434	1635	2151
5	305	0.39	15.967	17.623	5529	9092
6	59	0.38	1.453	1.601	608	997
7	6	0.38	1.126	1.195	85	107
8	81	0.39	3.653	4.539	1780	2406
9	480	0.38	23.376	35.117	9466	15756
10	555	0.39	31.382	47.217	11789	18822
11	109	0.40	5.231	5.625	2047	2913
12	428	0.38	17.275	24.976	7933	13363
13	559	0.38	20.525	43.574	9976	17741
14	52	0.38	2.031	2.626	745	941
15	77	0.39	3.973	4.821	1590	2137
16	77	0.39	3.896	4.633	1590	2137

Figure 3: **Comparison of our algorithm with Wroblewski's**

In this section, we present the actual results of experiments which compare our unification algorithm to Wroblewski's algorithm (slightly modified to handle variables and loops that are required by our HPSG based grammar). In the table, 'unifs' represents the total number of unifications during a parse (the number of calls to the top-level 'unify-dag', and not 'unify1'). 'USrate' represents the ratio of successful unifications to the total number of unifications. We parsed each sentence three times on a Symbolics 3620 using both unification methods and took the shortest elapsed time for both methods ('T' represents our scheme, 'W' represents Wroblewski's algorithm with a modification to handle loops and variables[12]). Data structures are the same for both unification algorithms (except for additional fields for a node in our algorithm, i.e., comp-arc-list, comp-arc-

[11] It may be slightly slower, because our unification recurses twice on a graph: once to unify and once to copy, whereas in incremental unification schemes copying is performed during the same recursion as unifying. Additional bookkeeping for incremental copying during unify2 may slightly offset this, however.

[12] Loops 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 the arc. ([Kogure, 1989]). Thus, loops 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 loops in input dags.

mark, and forward-mark). Same functions are used to interface with Earley's parser and the same subfunctions are used wherever possible (such as creation and access of arcs) to minimize the differences that are not purely algorithmic. 'Number of copies' represents the number of nodes created during each parse (and does not include the number of arc structures that are created during a parse).

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. In order to vary the number of unifications per parse (from 5 to over 500) as seen in the data, we used 16 sentences from a sample telephone conversation dialog which range from very short sentences (one word, i.e., *iie* 'no') to relatively long ones (such as *soredehakochirakarasochiranitourokuyoushiwoookuriitashimasu* 'In that case, we [speaker] will send you [hearer] the registration form.').

4. Discussion:

4.1. Comparison to Other Approaches

As we have seen the control structure of the algorithm is simple. Indeed it is essentially identical to that of [Pereira, 1985]. However, instead of storing changes to the argument dags in the environment we store the changes in the dags themselves non-destructively. Because we do not use the environment, the log(d) overhead (where d is the number of nodes in a dag) associated with Pereira's scheme that is required during node access (to assemble the whole dag from the skeleton and the updates in the environment) is avoided in our scheme. 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. Karttunen originally introduced this scheme in order to replace the less efficient structure-sharing implementations ([Pereira, 1985], [Karttunen and Kay, 1985]). The difference between Karttunen's method and ours is that in our algorithm, one increment to the global counter can invalidate all the changes made to nodes (the method that was used by Wroblewski to invalidate copy fields in his algorithm), while in Karttunen's algorithm each node in the entire argument graphs must be separately restored by retrieving and resetting the attribute-values saved in an array together with the original values. In both Karttunen's and our algorithm, there will be a non-destructive (reversible, and quasi-destructive) saving of intersection arcs that may be wasted when a subgraph of a particular node successfully unifies but the final unification fails due to a failure in some other part of the argument graphs. This is not a problem in our method because the temporary change made to a node is performed as pushing pointers into already existing structures (nodes) and it does not require entirely new structures to be created and dynamically allocated memory (which was necessary for the copy (create-node) operation).[13] [Godden, 1990] presents a method of using lazy evaluation in unification which seems to be one successful actualization of [Karttunen and Kay, 1985]'s lazy evaluation idea. One question about lazy evaluation is that the

[13] Although, in Karttunen's method it may be rather expensive if the array becomes large, because adding new values (i.e., attribute-values to be stored) to a long array can be expensive. (i.e, By storing the original values in an array, the saving operation becomes expensive in return for cheaper retrieval of the saved values – which we think is still expensive because retrieval needs to be done for each node in the input dags.)

efficiency of lazy evaluation varies depending upon the particular hardware and programming language environment. For example, in CommonLisp, to attain a lazy evaluation, as soon as a function is delayed, a closure (or a structure) needs to be created receiving a dynamic allocation of memory (just as in creating a copy node). Thus, there is a shift of memory and associated computation consumed from making copies to making closures. In terms of memory cells saved, although the lazy scheme may reduce the total number of copies created, if we consider the memory consumed to create closures, the saving may be significantly cancelled. In terms of speed, since delayed evaluation requires additional bookkeeping, how schemes such as the one introduced by [Godden, 1990] would compare with non-lazy incremental copying schemes is an open question. Unfortunately Godden offers a comparison of his algorithm with one that uses a full copying method (i.e. his Eager Copying) which is already significantly slower than Wroblewski's algorithm. However, no comparison is offered with prevailing unification schemes such as Wroblewski's. With the complexity for lazy evaluation and the memory consumed for delayed closures added, it is hard to estimate that lazy unification runs considerably faster than Wroblewski's incremental copying scheme. Finally, as we see in the next subsection, lazy unification does not seem effectively parallelizable.

5. Parallel Q-D Possibilities

5.1. Considerations for Parallelism

One potential method for speeding up unification would involve taking advantage of parallelism. We find the following two elements to be essential in parallelizing a graph unification algorithm.

- **A node should not be accessed concurrently by different processes.**
- **Parallelism should contribute significantly at recursive calls on shared arcs.**

Given that unification when it is parallelized is performed on a (global and) shared memory area, if more than one process tries to access the same node, i.e., same memory location, than the processes need to be queued (by lock/unlock on the memory location, or by blocking communication between processes). In other words, the operation becomes essentially sequential. Because there will be overhead associated with the lock/unlock and process communications, there will be little advantage in parallelizing if concurrent processes frequently access the same nodes (and arcs). This serializing of concurrent processes is a problem especially for write operations because multiple write operations performed simultaneously on the same location result in significant overhead (in addition to the serializing of concurrent processes). This seems to be a problem when we parallelize the incremental copying schemes (such as [Wroblewski, 1987] and [Godden, 1990]) because there may be multiple simultaneous write operations on a copy when recursive calls to the shared arcs at each level return successfully. Since non-determinism of recursive calls to the shared arcs is the source of the gain of parallelizing graph unification, serialization of parallel processes by simultaneous writes (and the resulting lock/unlock overhead) takes away the appeal for parallelizing graph unification. (The same problem will be caused during writes to 'copying environments' in the lazy unification scheme and saving of attribute-values into an array in reversible unification).

5.2. Strength of the Q-D algorithm for parallelism

Our algorithm does not suffer from this simultaneous write lock/unlock problem because there will be no write operation to a node during unification checks (i.e., no writing is performed until the unification of entire argument dags actually succeeds). In the parallel version, the quasi-destructive addition of intersection arcs to a node does not occur until all parallel recursive calls into subgraphs succeed. This can be performed without any harm because 1) any addition to the comp-arc-list is harmless until actual copying is performed after a successful unification; 2) additions to comp-arc-list are performed only once per node and therefore, this will not cause the lock/unlock problem due to multiple simultaneous write operations. However, the addition of temporary forwarding links needs to wait until the top-level unification successfully returns. This is because the unification at different levels of a graph continues simultaneously and therefore, the assumption that the forwarding of a node to another node means that the subgraph of the node has successfully unified, which is consistent with the serial implementation, no longer stands in the case of the parallel version. This means that if there is a convergent arc in the argument dag, for example in dag2, then in the serial version, the recursive unification for the subgraph below the convergent node is only performed once because there will be a temporary forwarding link as soon as the unification check succeeds first time around. In the parallel unification version, because we cannot create a temporary forwarding link during a unification, if there is a loop, the subgraph will be unified twice (by different processes). (Over copying is still avoided in such a case. It is just checked by two different processes simultaneously.) We have recently completed a preliminary implementation of the parallel version of our algorithm and in the near future we will be conducting experiments using the parallel version especially comparing it with the serial implementation and a straightforward parallel implementation of Wroblewski's algorithm[14] to study the actual performance increase due to parallelism.

6. Conclusion

The experiments show that the proposed Q-D algorithm runs significantly faster than Wroblewski's algorithm using Earley's parser and an HPSG based grammar developed at ATR. The gain comes from the fact that our algorithm creates no over copies and early copies at all. In Wroblewski's algorithm, although over copies are essentially avoided, early copies (by our definition) are a significant problem because about 60 percent of unifications result in failure in a successful parse in our sample parses. The additional set-difference operation, etc. required for incremental copying during unify2 may also be contributing to the slower speed of Wroblewski's algorithm. Given that our sample grammar is relatively small, we would expect that the difference in the performance between the incremental copying schemes and ours will expand as the grammar size increases and both the number of failures[15] and the size of the wasted subgraphs of failed unifications become larger. Since our algorithm is essentially parallel we are expecting a desirable gain by running it on a parallel machine. Parallel processes can be continuously created as unify1 recurses deeper and deeper without creating any copies by simply looking for a possible failure of the unification (and preparing for successive copying in case unification succeeds). So far, we have completed a preliminary implementation on a parallel hardware and we hope to collect data in the near future

[14] We hope to report the result of experiments using the parallel version at the Workshop.

[15] For example, in our large-scale speech-to-speech translation system under development, the USrate is estimated to be under 20%, i.e., over 80% of unifications are estimated to be failures.

using the parallel hardware. The proposed algorithm is desirable over existing algorithms first simply due to the demonstrated speed over currently fastest algorithm (Wroblewski's). Second, due to the simplicity of the control structure it should be easy to implement especially compared to the incremental copying algorithms. Currently our algorithm does not support structure sharing. In the near future we would like to first further test the robustness of the algorithm then we hope to propose a structure-sharing version of the algorithm for further improvement of the efficiency.

Appendix: Implementation

The programs are written in CommonLisp. Those were originally implemented on Symbolics and now runs on Sun Lucid and Allegro. For the experiments reported in this paper, the unification algorithms, Earley parser and the HPSG path equation to graph converter programs are run on Symbolics 3620. The preliminary parallel version of our unification algorithm is implemented on a Sequent Symmetry shared-memory parallel machine running Allegro CLiP parallel CommonLisp based on a micro-tasking parallelism using light-weight processes.

References

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

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

[Karttunen and Kay, 1985] Karttunen, L. and Kay, M. "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.

[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 Sag, A. *Information-based Syntax and Semantics.* Vol 1, CSLI, 1987.

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

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