Quasi-Destructive Graph Unification

Hideto Tomabechi

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

ATR Interpreting Telephony Research Laboratories*
Seika-cho, Sorakugun, Kyoto 619-02 JAPAN

ABSTRACT

Graph unification is the most expensive part of unification-based grammar parsing. It often takes over 90% of the total parsing time of a sentence. 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 elements without expensive overhead through temporarily modifying graphs during unification to eliminate copying during unification. We found that parsing relatively long sentences (requiring about 500 top-level unifications during a parse) using our algorithm is approximately twice as fast as parsing the same sentences using Wroblewski's algorithm.

1. Motivation

Graph unification is the most expensive part of unification-based grammar parsing systems. For example, in the three types of 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.[2] 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 (cycles 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 systems.

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 cycles[3] 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 the 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[4]. 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[5].

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 and space 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 per-

*Visiting Research Scientist. Local email address: tomabech%atr-la.atr.co.jp@uunet.UU.NET.

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

[2]In the large-scale HPSG-based spoken Japanese analysis system developed at ATR, sometimes 98 percent of the elapsed time is devoted to graph unification ([Kogure, 1990]).

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

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

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

cent of the 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 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[6]), 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.[7] 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 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 will be a significant problem when we consider the possibility of parallelizing the unification algorithms as well.

2. Our Scheme

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

As a data structure, a node is represented with 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

[6] 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.

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

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.

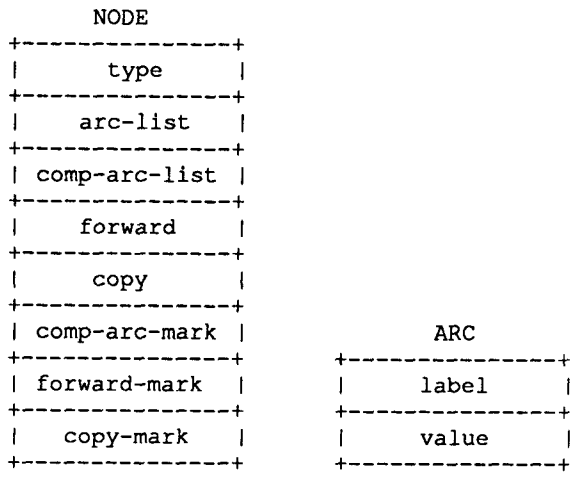

Figure 1: **Node and Arc Structures**

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 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[8]. As in [Pereira, 1985], we have three types of nodes: 1) :atomic, 2) :bottom[9], 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. Figure 2 is the central quasi-destructive graph unification algorithm and Figure 3 shows the algorithm for copying nodes and arcs (called by unify0) while respecting the contents of comp-arc-lists.

The functions Complementarcs(dg1,dg2) and Intersectarcs(dg1,dg2) are similar to Wroblewski's algorithm and 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) respectively. During the set-difference and set-intersection operations, the content of comp-arc-lists are respected as parts of arc lists if the comp-arc-marks match the current value of the global timing counter. Dereference-dg(dg) 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

[8]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.

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

value of *unify-global-counter*, then the function returns the forwarded node; otherwise it simply returns the input node. Forward(dg1, dg2, :forward-type) puts (the pointer to) dg2 in the forward field of dg1. If the keyword in the function call is :temporary, the current value of the *unify-global-counter* is written in the forward-mark field of dg1. If the keyword is :permanent, 9 is written in the forward-mark field of dg1. 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 dg feature-structures uses permanent forwarding to merge the additional grammatical specifications into a graph structure[10]. 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. 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 dg2 convergent arcs, let us consider the case when the destination node has a corresponding node in dg1 and only one of the convergent arcs has a corresponding arc in dg1. This destination node is already temporarily forwarded to the node in dg1 (since the unification check was successful prior to copying). Once a copy is created for the corresponding dg1 node and recorded in the copy field of dg1, every time a convergent arc in dg2 that needs to be copied points to its destination node, dereferencing the node returns the corresponding node in dg1 and since a copy of it already exists, this copy is returned. Thus no duplicate copy is created[11].

[10]We have been using Wroblewski's algorithm for the unification part of the parser and thus usage of (permanent) forwarding links is adopted by the grammar reader module to convert path equations to graphs. For example, permanent forwarding is done when a :bottom node is to be merged with other nodes.

[11]Copying of dg2 arcs happens for arcs that exist in dg2 but not in dg1 (i.e., Complementarcs(dg2,dg1)). Such arcs are pushed to the comp-arc-list of dg1 during unify1 and are copied into the arc-list of the copy during subsequent copying. If there is a cycle or a convergence in arcs in dg1 or in arcs in dg2 that do not have corresponding arcs in dg1, 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 dg1 or dg2.

[12]9 indicates a permanent forwarding link.

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

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

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

[16]This will be executed only when all recursive calls into unify1 succeeded. Otherwise, a failure would have caused

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 [12]
  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 = dg2)[13]THEN
     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)[14]THEN
           forward-dg(dg2,dg1,:temporary);
           return('*T*);
        ELSE throw[15]with keyword 'unify-fail;
  ELSE IF (dg1.type = :atomic OR
           dg2.type = :atomic) THEN
           throw with keyword 'unify-fail;
  ELSE new ← complementarcs(dg2,dg1);
       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);[16]
           dg1.comp-arc-mark ← *unify-global-counter*;
           dg1.comp-arc-list ← new;
           return ('*T*);
END;
```

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

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-mark = *unify-global-counter*) THEN
      return(dg.copy);[17]
  ELSE IF (dg.type = :atomic) THEN
      copy ← create-node();[18]
      copy.type ← :atomic;
      copy.arc-list ← dg.arc-list;
      dg.copy ← copy;
      dg.copy-mark ← *unify-global-counter*;
      return(copy);
  ELSE IF (dg.type = :bottom) THEN
      copy ← create-node();
      copy.type ← :bottom;
      dg.copy ← copy;
      dg.copy-mark ← *unify-global-counter*;
      return(copy);
  ELSE
      copy ← create-node();
      copy.type ← :complex;
      FOR ALL arc IN dg.arc-list DO
                newarc ← copy-arc-and-comp-arc(arc);
                push newarc into copy.arc-list;
      IF (dg.comp-arc-list is non-empty AND
          dg.comp-arc-mark = *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 copy.arc-list;
      dg.copy ← copy;
      dg.copy-mark ← *unify-global-counter*;
      return (copy);
END;

FUNCTION copy-arc-and-comp-arcs(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 3: **Node and Arc Copying Functions**

Figure 4 shows a simple example of quasi-destructive graph unification with dg2 convergent arcs. The round nodes indicate atomic nodes and the rectangular nodes indicate bottom (variable) nodes. First, top-level unify1 finds that each of the input graphs has arc-a and arc-b (*shared*). Then unify1 is recursively called. At step two, the recursion into arc-a locally succeeds, and a temporary forwarding link with time-stamp(n) is made from node []2 to node *s*. At the third step (recursion into arc-b), by the previous forwarding, node []2 already has the value *s* (by dereferencing). Then this unification returns a success and a temporary forwarding link with time-stamp(n) is created from node []1 to node *s*. At the fourth step, since all recursive unifications (unify1s) into shared arcs succeeded, top-level unify1 creates a temporary forwarding link with time-stamp(n) from dag2's root node to dag1's root node, and sets arc-c (*new*) into comp-arc-list of dag1 and returns success ('*T*). At the fifth step, a copy of dag1 is created respecting the content of comp-arc-list and dereferencing the valid forward links. This copy is returned as a result of unification. At the last step (step six), the global timing counter is incremented (n ⇒ n+1). After this operation, temporary forwarding links and comp-arc-lists with time-stamp (< n+1) will be ignored. Therefore, the original dag1 and dag2 are recovered in a constant time without a costly reversing operations. (Also, note that recursions into shared-arcs can be done in any order producing the same result).

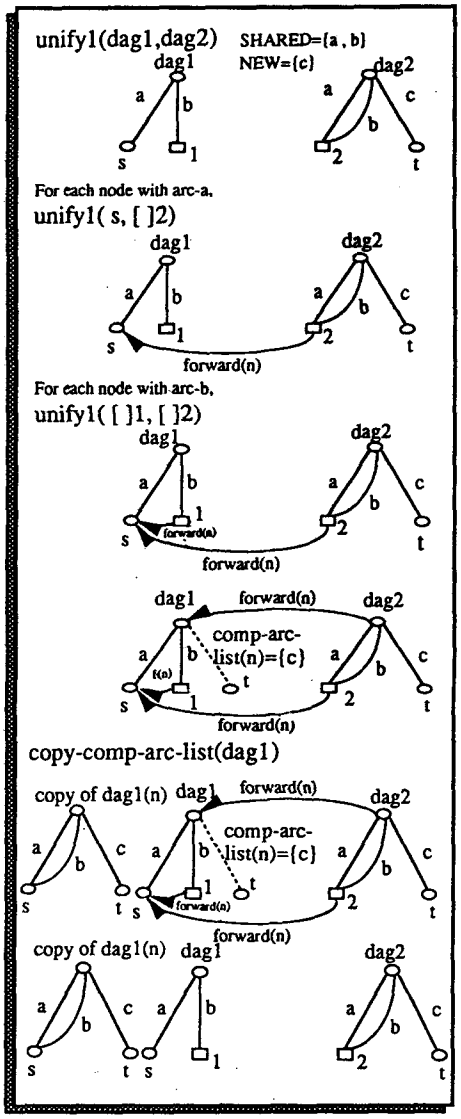

Figure 4: **A Simple Example of Quasi-Destructive Graph Unification**

an immediate return to *unify-dg*.

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

[18] Creates an empty node structure.

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(dg2,dg1) are set to the comp-arc-list of dg1 and not into the arc-list of dg1. Thus, as soon as we increment the global counter, the changes made to dg1 (i.e., addition of complement arcs into comp-arc-list) vanish. As long as the 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, dg1 is the result of unification. Hence the name quasi-destructive graph unification. In order to create a copy for subsequent use we only need to make a copy of dg1 before we increment the global counter while respecting the content of the comp-arc-list of dg1.

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 Wroblewski's algorithm[19]. 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. In the next section, we will see the actual results of experiments which compare our unification algorithm to Wroblewski's algorithm (slightly modified to handle variables and cycles that are required by our HPSG based grammar).

3. Experiments

Table 1 shows the results of our experiments using an HPSG-based Japanese grammar developed at ATR for a conference registration telephone dialogue domain. 'Unifs' represents the total number of unifications during a parse (the number of calls to the top-level 'unify-dg', 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 cycles and variables[20]). 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-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). 'Number of conses' represents the amount of structure words consed during a parse. This number represents the real comparison of the amount of space being consumed by each unification algorithm (including added fields for nodes in our algorithm and arcs that are created in both algorithms).

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. 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.'). Thus, the number of (top-level) unifications per sentence varied widely (from 6 to over 500).

[19] 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 and an additional set-difference operation (i.e, complementarcs(dg1,dg2)) during unify2 may offset this, however.

[20] 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 the arc. ([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.

sent#	Unifs	USrate	Elapsed time(sec)		Num of Copies		Num of Conses	
			T	W	T	W	T	W
1	6	0.5	1.066	1.113	85	107	1231	1451
2	101	0.35	1.897	2.899	1418	2285	15166	23836
3	24	0.33	1.206	1.290	129	220	1734	2644
4	71	0.41	3.349	4.102	1635	2151	17133	22943
5	305	0.39	12.151	17.309	5529	9092	57405	93035
6	59	0.38	1.254	1.601	608	997	6873	10763
7	6	0.38	1.016	1.030	85	107	1175	1395
8	81	0.39	3.499	4.452	1780	2406	18718	24978
9	480	0.38	18.402	34.653	9466	15756	96985	167211
10	555	0.39	26.933	47.224	11789	18822	119629	189997
11	109	0.40	4.592	5.433	2047	2913	21871	30531
12	428	0.38	13.728	24.350	7933	13363	81536	135808
13	559	0.38	15.480	42.357	9976	17741	102489	180169
14	52	0.38	1.977	2.410	745	941	8272	10292
15	77	0.39	3.574	4.688	1590	2137	16946	22416
16	77	0.39	3.658	4.431	1590	2137	16943	22413

Table 1: **Comparison of our algorithm with Wroblewski's**

4. Discussion: Comparison to Other Approaches

The control structure of our algorithm is 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]). In Karttunen's method[21], whenever a destructive change is about to be made, the attribute value pairs[22] 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.) 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, while in Karttunen's algorithm 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 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).[23] [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 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 canceled. 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 algo-

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

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

[23] Although, in Karttunen's method it may become rather expensive if the arrays require resizing during the saving operation of the subgraphs.

rithm 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 whether lazy unification runs considerably faster than Wroblewski's incremental copying scheme.[24]

5. Conclusion

The algorithm introduced in this paper 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 does not create any over copies or early copies. 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 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[25] and the size of the wasted subgraphs of failed unifications become larger. Since our algorithm is essentially parallel, parallelization is one logical choice to pursue further speedup. 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 shared memory parallel hardware with about 75 percent of effective parallelization rate. With the simplicity of our algorithm and the ease of implementing it (compared to both incremental copying schemes and lazy schemes), combined with the demonstrated speed of the algorithm, the algorithm could be a viable alternative to existing unification algorithms used in current natural language systems.

[24] That is, unless some new scheme for reducing excessive copying is introduced such as structure-sharing of an unchanged shared-forest ([Kogure, 1990]). Even then, our criticism of the cost of delaying evaluation would still be valid. Also, although different in methodology from the way suggested by Kogure for Wroblewski's algorithm, it is possible to attain structure-sharing of an unchanged forest in our scheme as well. We have already developed a preliminary version of such a scheme which is not discussed in this paper.

[25] 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.

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 and Jaime Carbonell at CMU. Thanks are also due to Margalit Zabludowski and Hiroaki Kitano for comments on the final version of this paper and Takako Fujioka for assistance in implementing the parallel version of the algorithm.

Appendix: Implementation

The unification algorithms, Earley parser and the HPSG path equation to graph converter programs are implemented in CommonLisp on a Symbolics machine. The preliminary parallel version of our unification algorithm is currently implemented on a Sequent/Symmetry closely-coupled shared-memory parallel machine running Allegro CLiP parallel CommonLisp.

References

[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.

[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.

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