Efficient Lazy Unification

Noriko Tomuro and Steven L. Lytinen
DePaul University
School of Computer Science, Telecommunications and Information Systems
243 S. Wabash Ave.
Chicago IL 60604
phone: (312)362-6106
fax: (312) 362-6116
cphdnt@ted.cs.depaul.edu

Abstract

Parsing with unification grammars is inefficient due to the intractable complexity of the algorithm. In implemented systems, performance suffers even more by additional overhead of processing large feature-value structures (FSs) as the base data structure. In particular, copying and unification of FSs has been identified to be the most expensive operation.

In this paper, we present an algorithm which copies FSs efficiently. The algorithm is a version of *lazy unification*, an optimization technique for unification-based systems. The algorithm utilizes a data structure called *environment* for bookkeeping, and it is relatively simple. The algorithm attains efficiency by *lazy evaluation*, in which FSs are copied only when necessary, and *structure sharing*, in which only the necessary portion of the FSs are copied. We present empirical testing which shows a marked effect by our algorithm in achieving a linear average case parsing performance.

Introduction

Unification grammar is a feature-based grammar formalism used in natural language processing. It enables the the grammar writer to account for syntactic phenomena which are difficult to capture in context-free grammars (e.g., subject-verb agreement) as well as many semantic phenomena. Unfortunately, parsing with unification grammars is inefficient due to the intractable complexity of the algorithm (Barton, *et el*, 1987; Rounds, 1991). In implemented systems, performance suffers even more from the additional overhead of processing large feature-value structures (FSs) such as directed acyclic graphs (DAGs) as the base data structure. In particular, since unification is most often implemented as a destructive operation, when one FS is unified with another FS more than once, it must be copied in order to avoid interference from later unification with other FSs. Copying and unification of large FSs has been reported to be the most expensive operations (Tomabechi, 1992; Godden, 1990).

The most naive approach to avoiding destructive changes made during unification is to copy the FS arguments in their entirety prior to the unification. However, this approach is inefficient in several ways. First, when unification fails, copies are wasted (*Early-copying* problem). Second, nodes in the intersection of two FSs are copied twice (*Over-copying* problem). Third, unmodified portions of the original FS's may not have to be copied immediately (i.e., sub-structures can be *shared* by multiple FSs) (*Redundant-copying* problem).

To remedy those problems, we developed an algorithm which copies FSs more efficiently. Our algorithm is a version of *lazy unification* (Godden, 1990; Tomabechi, 1992). Our algorithm utilizes a data structure called an *environment*[1] for bookkeeping purposes, and it is relatively simple and straight-forward. This algorithm has been implemented and incorporated in a unification-based parser called LINK (Lytinen, 1992). By using this efficient algorithm, LINK showed a significant reduction in the number of DAGs and CPU time used, and achieved linear performance in parsing with a unification grammar.

The Unification Operation

Unification grammar rules can be thought of themselves as FS's implemented as DAGs. Figure 1 shows an example LINK grammar rule. In our system, the root of the DAG for a grammar rule must have a set of numbered arcs, labelled {1, 2, ..., n}, with n corresponding to the arity of the rule. These arcs correspond to the phrase structure component of a rule. In addition, the root typically (but not always) has a **cat** feature, a syntactic category which corresponds to the left hand side of a context-free production. Nodes pointed to by numbered arcs also typically have **cat** arcs. These correspond to the right hand side of a context-free production. For example, the example rule in figure 1 in the diagram corresponds to the context-free rule **S** $\rightarrow$ **NP VP**. Additional features further constrain the constituents involved in a rule. For example, in the **S** rule, the **NP** and **VP** must have agreement features (**agr**) which are compatible.

[1] Our *environment* is slightly different from the *environment* field in a *molecule* (representation for a DAG node) used in (Pereira, 1985). The difference is described in **related work** section.

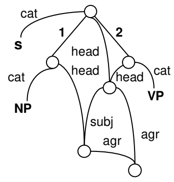

Figure 1: Example LINK grammar rule

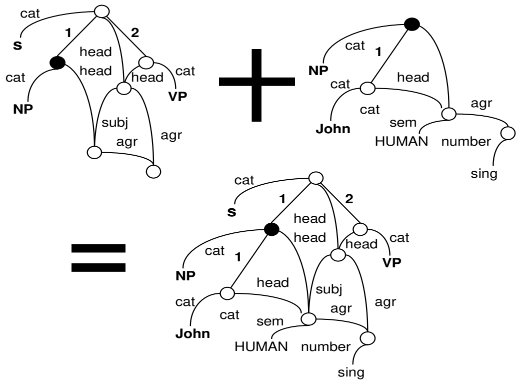

Figure 2: Example of unification in parsing

A new FS may be constructed by the parser by matching the numbered arcs of a grammar rule DAG with other FS's that have already been constructed from the input sentence. The matching operation is referred to as *unification*. The result of successfully unifying two DAGs is a DAG with all features from both of the original DAGs. Two DAGs fail to unify if they share a feature with incompatible values. Figure 2 shows an example unification operation, in which a **NP** constructed from the word "John" is unified with the subDAG pointed to by the **1** arc of the DAG in figure 1.

Because English is often locally (or even globally) ambiguous, it is frequently necessary for the parser to construct multiple FS's corresponding to several possible interpretations of a sentence fragment (or a complete sentence). Since unification is a destructive operation, this means that FS's must often be copied, in order to prevent interference with representations of alternative interpretations. The efficiency of this copying operation is thus central to the overall efficiency of unification-based parsing.

Lazy Unification

The idea of *lazy unification* is to delay copying a DAG until destructive modification is about to happen to the DAG. A DAG in LINK consists of a set of nodes[2] which may be labelled, and a set of edges or arcs (also labelled) which connect nodes. A DAG is modified when a change is made to either the label or any of the outgoing arcs, or both. Sometimes unification does not require any modification at all, such as when one DAG properly subsumes the other DAG. In those cases, the more specific of the two DAGs can be returned as the result of unification without making copies. For example, when D1 and D2 in Figure 3[3] are unified, D1 can be simply returned as the result of unification. If any modification is needed for D1 later in the parsing, it is copied at that time (*lazy evaluation*).

[2]A single node may also be thought of as a DAG with that node as its root; hence, we will sometimes refer to nodes as DAGs.

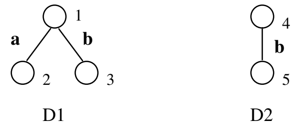

Figure 3: Example of lazy unification (1)

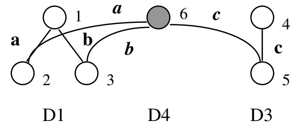

Figure 4: Example of lazy unification (2)

When unification requires modifications to a DAG, only the nodes being modified are copied. Nodes which are not modified can remain uncopied, and will be *shared* by the original DAG(s) and the result DAG (*Structure Sharing*). For example, when D1 and D3 in Figure 4[4] are unified, modification is needed for the top node of D1 (node 1[5]) since it must have an additional arc labeled **c** to node 5. For other nodes (node 2, 3 and 5), no modification is required. Therefore, node 1 is copied (thereby creating node 6) that will then have the arcs to all the nodes which node 1 and node 4 point to. Since there is no other node which needs to be copied, a DAG rooted by node 6 (D4) is returned as the result of unifying D1 and D3. Notice D4 (result DAG) now shares nodes with D1 and D3 (original DAGs) without interference.

[3]In all Figures, node number is indicated next to the node symbol (a circle). Node labels are omitted for brevity: labels of all unified nodes are assumed to unify.

[4]In all Figures, copied nodes are indicated by shaded circles, and arc labels from the copied nodes are italicized.

[5]In our algorithm, when a node is in the overlapping portion of two DAGs (*dag1* and *dag2*) being unified, the one in *dag1* is used as the original node.

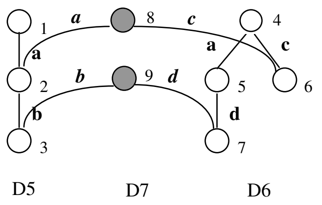

Figure 5: Example of lazy unification (3)

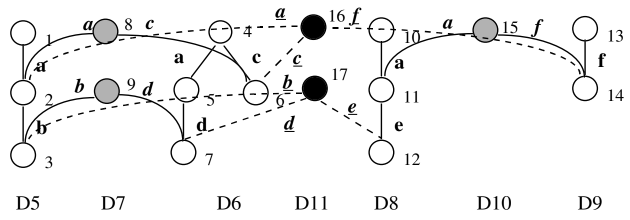

Figure 6: Merging Environments

Copy Environment

The lazy unification in LINK utilizes a globally defined data structure called *environment*. The environment is a table which records a *copy history* for all root nodes of the DAGs created during the parse. A copy history of a DAG is a list of associations between the original and the copy of the nodes (subDAGs) within the DAG. Whenever a node in a DAG is copied, an association of two nodes, in the form of original-copy pair, is entered in the copy history of the root node of the DAG, and this information is later used to dereference the node.[6] In other words, every root node carries a context in which all its subDAGs are evaluated.

For example, when D5 and D6 in Figure 5 are unified, node 1 and node 2 are copied (thereby creating node 8 and node 9 respectively): node 8 will have an arc labeled **a** to node 2, plus another arc labeled **c** to node 6; node 9 will have an arc labeled **b** to node 3, plus another arc labeled **d** to node 7. After the unification, the environment table will have the copy history for node 8 (root node of the result DAG D7):

```
#<node 8> ((1 8) (2 9))
```

Copy histories in the environment table are used to dereference nodes in different contexts. For example, when node 2 is accessed within D5 (following the arc labeled **a** from node 1), no dereference takes place since there is no copy history in the environment table for node 1 (i.e., all nodes in D5 are original). However, when node 2 is accessed within D7 (following the arc labeled **a** from node 8), it is dereferenced to node 9 since the pair (2 9) is recorded in the copy history in the environment table for node 8. This way, a node is accessed by following the arc from the parent to the original destination node first, and then the environment table is used to correctly dereference the node in the context of a particular DAG in which it is evaluated. Then, a DAG is reconstructed by traversing the arcs directly indicated in the nodes, and looking up the original-copy associations recorded in the copy history of the DAG.

Note that, since a DAG node has a recursive structure, it can be considered as the root of the DAG of itself. If every node is thought of as a root node, the environment table could possibly become very large. However, a root node in our notion refers to the top node of a DAG which represents some linguistic object such as grammar rule or word entry. So, the root corresponds to the DAG node on which top-level unification is called. Since there are limited number of such nodes, the size of the environment table will not become as large.

When two DAGs are unified, the resulting DAG must inherit the contexts of both original DAGs in order to ensure the correctness of node dereferencing. Suppose unification is called on D7 and D10 in Figure 6[7] (D10 is the result of unifying D8 and D9). Initially, the environment table contains copy histories:

```
#<node 8> ((1 8) (2 9))
#<node 15> ((10 15))
```

During unification, node 8 is first copied (since it will need to have an additional arc labeled **f** to node 14), and node 16 is created. Since this node will be the root of the result DAG, a new copy history for node 16 is obtained by merging the copy histories of node 8 and node 15. After the original-copy association between node 8 and 15 is added in the copy history, the environment table now contains the following copy histories:

```
#<node 8> ((1 8) (2 9))
#<node 15> ((10 15))
#<node 16> ((1 8) (2 9) (10 15) (8 16))
```

After three arcs are set for node 16 (one arc labeled **a** to node 2, another one labeled **c** to node 6, and one more arc labeled **f** to node 14), node 9[8] is copied next (since it will need to have an additional arc labeled **e** to node 12), and node 17 is created. At this time, the association between 9 and 17 is added in the copy history of node 16 (root node of the result DAG 11). Now the environment table will contain:

[6] Since a node can be shared by multiple DAGs, derefrencing may return different nodes depending on the root node.

[7] Nodes that were copied by previous unifications are indicated with lightly-shaded circles, and the nodes in the result DAG (D11) are indicated with heavily-shaded circles. Arcs from node 16 are indicated by dashed lines, and their labels are italicized and underlined.

[8] Node 9 is obtained by dereferencing node 2 in the context of D7.

```
FUNCTION unify-dags(dag1, dag2, root1, root2)
  d1 <- deref(dag1, root1)
  d2 <- deref(dag2, root2)
  IF (d1 = d2) THEN
    return (d1)
  ELSE IF (d2 not nil) THEN
    unify-within(d1, d2, root1)
  ELSE
    new-label <- unify-labels(d1.label, d2.label)
    IF (new-label not nil) THEN
      add (d2 . d1) in *unify2-table*
      shared-arcs <- intersection(d1.arcs, d2.arcs)
      comp-arcs <- set-difference(d2.arcs, shared-arcs)
      IF (new-label != d1.label OR
          comp-arcs not empty) THEN
        new-node <- copy-dag-node(d1, root1)
        new-node.label = new-label
        new-node.arcs = union(d1.arcs, d2.arcs)
        FOR EACH arc IN shared-arcs DO
          unify-dags(destination of arc from new-node,
                     destination of arc from d2,
                     root1, root2)
        return (new-node)
      ELSE
        FOR EACH arc IN shared-arcs DO
          unify-dags(destination of arc from d1,
                     destination of arc from d2,
                     root1, root2)
        return (d1)
      ENDIF
    ENDIF
  ENDIF
END.
```

Figure 7: Lazy Unification Algorithm

```
#<node 8> ((1 8) (2 9))
#<node 15> ((10 15))
#<node 16> ((1 8) (2 9) (10 15) (8 16) (9 17))
```

Notice a chain of original-copy associations has been formed for node 2 in D11: from node 2 to node 9 to node 17. When node 2 is accessed within D11 later in the parsing, this chain of associations is traversed and node 17 (last copy node in the context of D11) is returned.

The Lazy Unification Algorithm

The algorithm for our lazy unification is rather simple and straight-forward. It is essentially tail-recursive, and incrementally copies the DAG nodes without backward propagation.

We shall assume that prior to the call to the unification function, the root node of the first argument DAG (*dag1*) is copied, and the initial copy history for the node (i.e., root of the result DAG) is already entered in the environment table.

Figure 7 shows the rough description of our lazy unification algorithm. The function *deref* dereferences the DAG node with respect to the root node in the environment table. This function also utilizes **unify2-table** which records reference links from nodes in *dag2* to nodes in *dag1*. The function *unify-within* handles the case when nodes in *dag1* that have been processed already in the previous recursive calls are made re-entrant by *dag2*. The function *unify-labels* checks

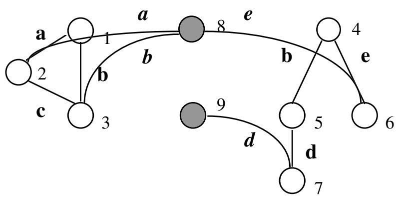

Figure 8: Example of lazy unification (4)

whether two labels are compatible.[9] The function *copy-dag-node* copies the argument DAG node, and adds the original-copy association in the copy history of the argument root node in the environement table.

Discussion

Our lazy unification algorithm copies DAGs efficiently: by *lazy evaluation* scheme, DAGs are copied only when necessary; by *structure sharing* scheme, only the neccesary portion of the DAGs are copied. Therefore, our algorithm solves over-copying and redundant-copying problems, and reduces the number of DAGs created during the parsing.

Our lazy unification algorithm is very effective particularly when DAGs have re-entrant structure. Suppose DAGs D12 and D13 shown in Figure 8 are unified. In the first call to *unify-dags*, node 1 is copied since it will need to have an additional arc labeled **e** to node 6. In the next recursive call after traversing the arc labeled **b**, node 3 is copied since it will need to have an additional arc labeled **d** to node 7. At this point, since there are no more shared arcs from node 9 or node 5, unification terminates.

Notice node 2 was not copied during unification. In our algorithm, even when the arc destination node is copied, the parent node need not be copied. This is because nodes in a DAG do not have to be connected by direct arcs: original nodes are accessed by direct arcs, while copied nodes are accessed through the original-copy associations in the copy history. If only direct arcs were used, if an update is found for the nodes in the subDAG that is accessible through more than one path, this information must be propagated upward, thus requiring re-processing of the previous recursive unification calls. In order to avoid such redundant processing, special dependency information must be maintained. However, this scheme adds an extra overhead, and makes the algorithm more complex. Our algorithm solves this problem by allowing disconnected DAGs which are reconstructed using the copy environment table. This scheme enables a one-pass,

[9] In LINK, every DAG label corresponds to a concept in a separately defined semantic net (knowledge base). So, two labels are considered compatible if they have a IS-A relation in the semantic hierarchy.

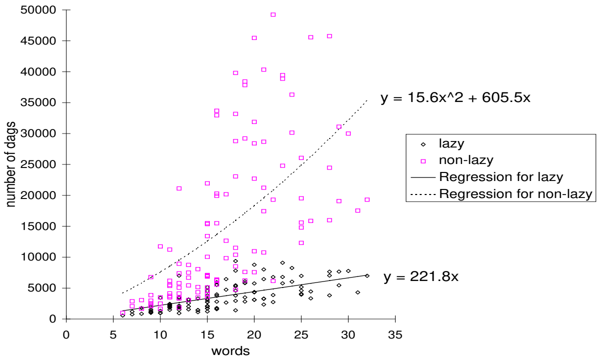

Figure 9: Number of DAGs vs. sentence length

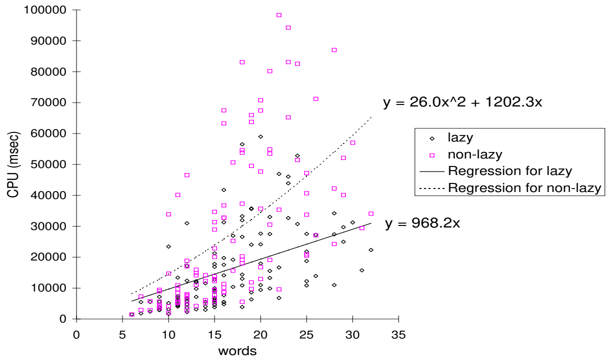

Figure 10: CPU time vs. sentence length

tail-recursive algorithm with no upward propagation, thereby simplifying the complexity of the algorithm.

One of the difficulties of our algorithm is the overhead of tracing the chain of associations in the copy history to dereference a node. Since copy histories are merged by unification, the length of a copy history will become longer as the parsing proceeds. However, the depth of a chain in a copy history does not grow so much in practice, because copied nodes as well as original nodes are shared as the parsing proceeds further. Therefore, the cost of accessing a node can remain relatively small throughout the parsing.

Empirical Result

Our lazy unification algorithm has been implemented and successfully incorporated in LINK. LINK uses a left-corner parsing algorithm which utilizes all available information including syntactic, semantic and domain knowledge in top-down filtering, and has previously achieved a quadratic average case performance for limited domain texts (Lytinen and Tomuro, 1996).

To evaluate the efficiency of our lazy unification algorithm, we selected a test corpus consisting of 113 sentences taken randomly from the Fifth Message Understanding Competition (MUC-5) (Sundheim, 1993), and parsed them using LINK. To compare the performance of lazy unification, the test was run on two versions of LINK[10]: one with lazy unification (the current system; henceforth referred to as the lazy version), and the other without lazy unification (the previous system; henceforth referred to as the non-lazy version). Their performance was compared in terms of number of DAGs and CPU time. Our hypothesis was that, by using lazy unification, the number of DAGs created during the parsing might decrease significantly enough so that the system would achieve a linear parsing performance.

[10] Since lazy unification does not affect the correctness of the unification operation, the same set of 113 sentences were parsed by both versions.

With respect to the number of DAGs, the lazy version achieved an average 75% reduction compared to the non-lazy version. We speculate that the biggest contribution was from the terminal nodes: since terminal nodes have no outgoing arcs, and their labels are often unchanged, they are shared by many DAGs in the lazy version. Figure 9 shows the plot of the number of DAGs vs. sentence length of the two versions. A best regression model analysis,[11] using the adjusted R^2 metric, indicates the number of DAGs created is linearly proportional to the sentence length by the lazy version (adj $R^2 = .856$), whereas a quadratically proportional number of DAGs were created by the non-lazy version (adj $R^2 = .738$). Assuming it takes a constant time to create a single DAG, this result strongly supported our hypothesis that lazy unification may bring the linear parsing performance.

With respect to CPU time, the lazy version achieved an average 44% reduction compared to the non-lazy version.[12] Figure 10 shows plots of CPU time vs. sentence length of the two versions. A best regression model analysis[13] indicates the lazy version indeed achieved a linear performance (adj $R^2 = .710$), compared to the quadratic model for the non-lazy version (adj $R^2 = .705$).

[11] In all analyses, best-fitting models were restricted to those with no constant coefficient (i.e., only curves which pass through the origin). This is because when analyzing the number of DAGs, since parsing a sentence containing 0 words requires no DAGs.

[12] We speculate that the difference of the reduction ratio between the number of DAGs and CPU time comes from the processing overhead by other parts of the system, such as the time to retrieve applicable rules.

[13] Intuitively, the best model of CPU time vs. sentence length may contain a constant coefficient, since the algorithm may include some constant-time components; however, when allowing for a constant coefficient, the best regression model results in a negative value for the constant. Thus, we did not allow for constant coefficients in the best models.

Related Work

In the unification grammar research paradigm, several efficient unification algorithms have been proposed. *Non-destructive Unification* proposed in (Wroblewski, 1987) copies nodes incrementally as the DAGs are traversed during unification. Although this approach solves over-copying problem, it does not support *structure-sharing*.

The approach taken in (Pereira, 1985) supports structure-sharing by having the result DAG share some nodes with the original DAGs. In this approach, a DAG is represented as a *molecule*, a pair consisting of a *skelton* structure and an *update environment*. The skelton structure keeps the features in the original structure, and modifications made by unification are recorded in the environment. However, since the environment is maintained in an array, every node access takes $O(lgn)$ time where n is the number of nodes in a DAG.

In (Godden, 1990), *Lazy Unification* with delayed *closure* is proposed. In this approach, when a node is first accessed during one unification, a closure object is created for the node. When the closure is accessed again during another unification, it is fully copied. Although this approach employs lazy evaluation scheme, closure objects in essense only substitute the copied nodes temporarily; therefore laziness is not truely attained.

In (Tomabechi, 1992), *Quasi-destructive Unification with structure-sharing* is proposed. Not only does this approach solve over-copying and redundant-copying problems by structure-sharing, it also solves early-copying problem. In order to eliminate wasted copies in the case of unification failure, this approach makes two passes over the structure arguments: the first pass determines if the unification will succeed or fail while recording the modifications in the temporary slot in the nodes; if unification is going to succeed, the second pass constructs the result structure by using the information kept in the temporary slot. Although this approach solves all three problems, it has some difficulties. First, the cost of making two passes pays off only if unification failure occurs significantly more often than success. Second, to support the two pass scheme, extra bookkeeping is required; therefore the algorithm necessarily become more complex. Our lazy unification algorithm tries to gain efficiency by less processing overhead and reduced algorithm complexity.

Conclusion

Lazy unification is an optimization technique for unificaton-based systems. In general, lazy unification requires a complex algorithm and data structures for bookkeeping, therefore adding an extra processing overhead. In most cases, the cost incurred by additional bookkeeping is conpensated by a significant reduction of the processing time to produce (decreased number of) FSs. Indeed, our experiment showed a marked effect by our lazy unification in achieving a linear average case performance in LINK.

Our lazy unification algorithm accomplishes the optimization by simple mechanisms: record modifications in the environment table, and use that information to dynamically construct a DAG. By allowing disconnected DAGs, our algorithm avoids the additional complexity needed to propagate modifications back to previous recursive calls. As a result, we have a less complex algorithm which unifies DAGs efficiently.

References

Barton, G., R. Berwick and E. Ristad (1987). *Computational complexity and natural language*, Cambridge, MA: MIT Press.

Godden, K. (1990). Lazy unification, in *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, Pittsburgh, PA, pp. 180-187.

Lytinen, S. (1992). A unification-based, integrated natural language processing system. *Computers and Mathematics with Applications, 23*(6-9), pp. 403-418.

Lytinen, S. and Tomuro, N. (1996). Left-corner Unification-based Natural Language Processing, in *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, OR, pp. 1037-1043.

Pereira, F. (1985). A structure-sharing representation for unification-based grammar formalisms, in *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, Chicago, IL, pp. 137-144.

Rounds, W. (1991). The relevance of computational complexity theory to natural language processing, in *Foundational Issues in Natural Language Processing*, Peter Sells, S. Shieber and T. Wasow (eds.), Cambridge, MA: MIT Press.

Sundheim, B. (1993). *Proceedings of the Fifth Message Understanding Conference (MUC-5).* San Francisco: Morgan Kaufmann Publishers.

Tomabechi, H. (1992). Quasi-destructive graph unification with structure-sharing, in *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92)*, Nantes, Italy, pp. 440-446.

Wroblewski, D. (1987). Non-destructive graph unification, in *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, pp. 582-587.