UNIFICATION WITH LAZY NON-REDUNDANT COPYING

Martin C. Emele*
Project Polygloss
University of Stuttgart
IMS-CL/IfI-AIS, Keplerstraße 17
D 7000 Stuttgart 1, FRG
emele@informatik.uni-stuttgart.de

Abstract

This paper presents a unification procedure which eliminates the redundant copying of structures by using a lazy incremental copying approach to achieve structure sharing. Copying of structures accounts for a considerable amount of the total processing time. Several methods have been proposed to minimize the amount of necessary copying. Lazy Incremental Copying (LIC) is presented as a new solution to the copying problem. It synthesizes ideas of lazy copying with the notion of chronological dereferencing for achieving a high amount of structure sharing.

Introduction

Many modern linguistic theories are using feature structures (FS) to describe linguistic objects representing phonological, syntactic, and semantic properties. These feature structures are specified in terms of constraints which they must satisfy. It seems to be useful to maintain the distinction between the constraint language in which feature structure constraints are expressed, and the structures that satisfy these constraints. Unification is the primary operation for determining the satisfiability of conjunctions of equality constraints. The efficiency of this operation is thus crucial for the overall efficiency of any system that uses feature structures.

Typed Feature Structure Unification

In unification-based grammar formalisms, unification is the meet operation on the meet semi-lattice formed by partially ordering the set of feature structures by a subsumption relation [Shieber 86]. Following ideas presented by [Aït-Kaci 84] and introduced, for example, in the unification-based formalism underlying HPSG [Pollard and Sag 87], first-order unification is extended to the sorted case using an order-sorted signature instead of a flat one.

In most existing implementations, descriptions of feature structure constraints are not directly used as models that satisfy these constraints; instead, they are represented by directed graphs (DG) serving as satisfying models. In particular, in the case where we are dealing only with conjunctions of equality constraints, efficient graph unification algorithms exist. The graph unification algorithm presented by Aït-Kaci is a node merging process using the UNION/FIND method (originally used for testing the equivalence of finite automata [Hopcroft/Karp 71]). It has its analogue in the unification algorithm for rational terms based on a fast procedure for congruence closure [Huet 76].

Node merging is a destructive operation

Since actual merging of nodes to build new node equivalence classes modifies the argument DGs, they must be copied before unification is invoked if the argument DGs need to be preserved. For example, during parsing there are two kinds of representations that must be preserved: first, lexical entries and rules must be preserved. They need to be copied first before a destructive unification operation can be applied to combine categories to form new ones; and second, nondeterminism in parsing requires the preservation of intermediate representations that might be used later when the parser comes back to a choice point to try some yet unexplored options.

*Research reported in this paper is partly supported by the German Ministry of Research and Technology (BMFT, Bundesminister für Forschung und Technologie), under grant No. 08 B3116 3. The views and conclusions contained herein are those of the authors and should not be interpreted as representing official policies.

DG copying as a source of inefficiency

Previous research on unification, in particular on graph unification [Karttunen/Kay 85, Pereira 85], and others, identified *DG copying* as the main source of inefficiency. The high cost in terms of time spent for copying and in terms of space required for the copies themselves accounts for a significant amount of the total processing time. Actually, more time is spent for copying than for unification itself. Hence, it is crucial to reduce the amount of copying, both in terms of the number and of the size of copies, in order to improve the efficiency of unification.

A naive implementation of unification would copy the arguments even before unification starts. That is what [Wroblewski 87] calls *early copying*. Early copying is wasted effort in cases of failure. He also introduced the notion of *over copying*, which results from copying both arguments in their entirety. Since unification produces its result by merging the two arguments, the result usually contains significantly fewer nodes than the sum of the nodes of the argument DGs.

Incremental Copying

Wroblewski's nondestructive graph unification with incremental copying eliminates early copying and avoids over copying. His method produces the resulting DG by incrementally copying the argument DGs. An additional copy field in the DG structure is used to associate temporary forwarding relationships to copied nodes. Only those copies are destructively modified. Finally, the copy of the newly constructed root will be returned in case of success, and all the copy pointers will be invalidated in constant time by incrementing a global generation counter without traversing the arguments again, leaving the arguments unchanged.

Redundant Copying

A problem arises with Wroblewski's account, because the resulting DG consists only of newly created structures even if parts of the input DGs that are not changed could be shared with the resultant DG. A better method would avoid (eliminate) such *redundant* copying as it is called by [Kogure 90].

Structure Sharing

The concept of structure sharing has been introduced to minimize the amount of copying by allowing DGs to share common parts of their structure.

The Boyer and Moore approach uses a skeleton/environment representation for structure sharing. The basic idea of structure sharing presented by [Pereira 85], namely that an initial object together with a list of updates contains the same information as the object that results from applying the updates destructively to the initial object, uses a variant of Boyer and Moore's approach for structure sharing of term structures [Boyer/Moore 72]. The method uses a skeleton for representing the initial DG that will never change and an environment for representing updates to the skeleton. There are two kinds of updates: *reroutings* that forward one DG node to another; *arc bindings* that add to a node a new arc.

Lazy Copying as another method to achieve structure sharing is based on the idea of lazy evaluation. Copying is delayed until a destructive change is about to happen. *Lazy copying* to achieve structure sharing has been suggested by [Karttunen/Kay 85], and lately again by [Godden 90] and [Kogure 90].

Neither of these methods fully avoids redundant copying in cases when we have to copy a node that is not the root. In general, all nodes along the path leading from the root to the site of an update need to be copied as well, even if they are not affected by this particular unification step, and hence could be shared with the resultant DG. Such cases seem to be ubiquitous in unification-based parsing since the equality constraints of phrase structure rules lead to the unification of substructures associated with the immediate daughter and mother categories. With respect to the overall structure that serves as the result of a parse, these unifications of substructures are even further embedded, yielding a considerable amount of copying that should be avoided.

All of these methods require the copying of arcs to a certain extent, either in the form of new arc bindings or by copying arcs for the resultant DG.

Lazy Incremental Copying

We now present Lazy Incremental Copying (LIC) as a new approach to the copying problem. The method is based on Wroblewski's idea of incrementally producing the resultant DG while unification proceeds, making changes only to copies and leaving argument DGs untouched. Copies are associated with nodes of the argument DGs by means

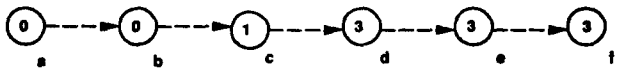

Figure 1: Chronological dereferencing.

of an additional copy field for the data structures representing nodes. But instead of creating copies for all of the nodes of the resultant DG, copying is done *lazily*. Copying is required only in cases where an update to an initial node leads to a destructive change.

The Lazy Incremental Copying method constitutes a synthesis of Pereira's structure sharing approach and Wroblewski's incremental copying procedure combining the advantages and avoiding disadvantages of both methods. The structure sharing scheme is imported into Wroblewski's method eliminating redundant copying. Instead of using a global branch environment as in Pereira's approach, each node records it's own updates by means of the copy field and a generation counter. The advantages are a uniform unification procedure which makes complex merging of environments obsolete and which can be furthermore easily extended to handle disjunctive constraints.

Data Structures

CopyNode structure	
type:	<symbol>
arcs:	<a list of ARCs>
copy:	<a pointer to a CopyNode>
generation:	<an integer>

ARC structure	
label:	<symbol>
dest:	<a CopyNode>

Dereferencing

The main difference between standard unification algorithms and LIC is the treatment of dereference pointers for representing node equivalence classes. The usual dereferencing operation follows a possible pointer chain until the class representative is found, whereas in LIC dereferencing is performed according to the current environment. Each copynode carries a generation counter that indicates to which generation it belongs. This means that every node is connected with its derivational context. A branch *environment* is represented as a sequence of valid generation counters (which could be extended to trees of generations for representing local disjunctions). The *current* generation is defined by the last element in this sequence. A copynode is said to be an *active* node if it was created within the current generation.

Nondeterminism during parsing or during the processs of checking the satisfiability of constraints is handled through chronological backtracking, i.e. in case of failure the latest remaining choice is reexamined first. Whenever we encounter a choice point, the environment will be extended. The length of the environment corresponds to the number of stacked choice points. For every choice point with n possible continuations, $n-1$ new generation counters are created. The last alternative pops the last element from the environment, continues with the old environment and produces n DG representations, one for each alternative. By going back to previous generations, already existing nodes become active nodes, and are thus modified destructively. This technique resembles the last call optimization technique of some Prolog implementations, e.g. for the WAM [Warren83]. The history of making choices is reflected by the dereferencing chain for each node which participated in different unifications.

Figure 1 is an example which illustrates how dereferencing works with respect to the environment: node `b` is the class representative for environment `<0>`, node `c` is the result of dereferencing for environments `<0 1>` and `<0 1 2>`, and finally node `f` corresponds to the representative for the environment `<0 1 2 3>` and all further extensions that did not add a new forwarding pointer to newly created copynodes.

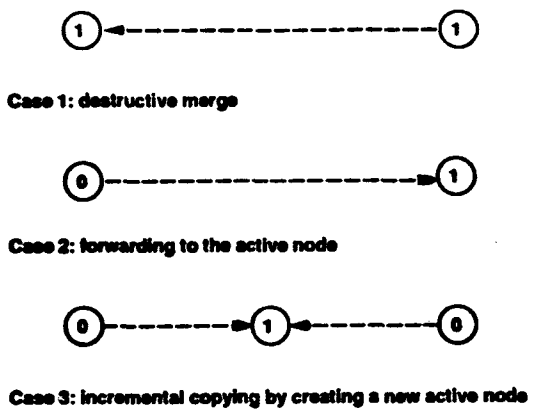

Figure 2: Node merging.

Advantages of this new dereferencing scheme are:

- It is very easy to undo the effects of unification upon backtracking. Instead of using trail information which records the nodes that must be restored in case of returning to a previous choice point, the state of computation at that choice point is recovered in constant time by activating the environment which is associated with that choice point. Dereferencing with respect to the environment will assure that the valid class representative will always be found. Pointers to nodes that do not belong to the current environment are ignored.
- It is no longer necessary to distinguish between the forward and copy slot for representing permanent and temporary relationships as it was needed in Wroblewski's algorithm. One copy field for storing the copy pointer is sufficient, thus reducing the size of node structures. Whether a unification leads to a destructive change by performing a rerouting that can not be undone, or to a nondestructive update by rerouting to a copynode that belongs to a higher generation, is reexpressed by means of the environment.

Lazy Non-redundant Copying

Unification itself proceeds roughly like a standard destructive graph unification algorithm that has been adapted to the order-sorted case. The distinction between *active* and *non-active* nodes allows us to perform copying lazily and to eliminate redundant copying completely.

Recall that a node is an active one if it belongs to the current generation. We distinguish between three cases when we merge two nodes by unifying them: (i) both are active nodes, (ii) either one of them is active, or (iii) they are both non-active. In the first case, we yield a destructive merge according to the current generation. No copying has to be done. If either one of the two nodes is active, the non-active node is forwarded to the active one. Again, no copying is required. When we reset computation to a previous state where the non-active node is reactivated, this pointer is ignored. In the third case, if there is no active node yet, we know that a destructive change to an environment that must be preserved could occur by building the new equivalence class. Instead, a new copynode will be created under the current active generation and both nodes will be forwarded to the new copy. (For illustration cf. Figure 2.) Notice that it is not necessary to copy arcs for the method presented here. Instead of collecting all arcs while dereferencing nodes, they are just carried over to new copynodes without any modification. That is done as an optimization to speed up the computation of arcs that occur in both argument nodes to be unified (`SharedArcs`) and the arcs that are unique with respect to each other (`UniqueArcs`).

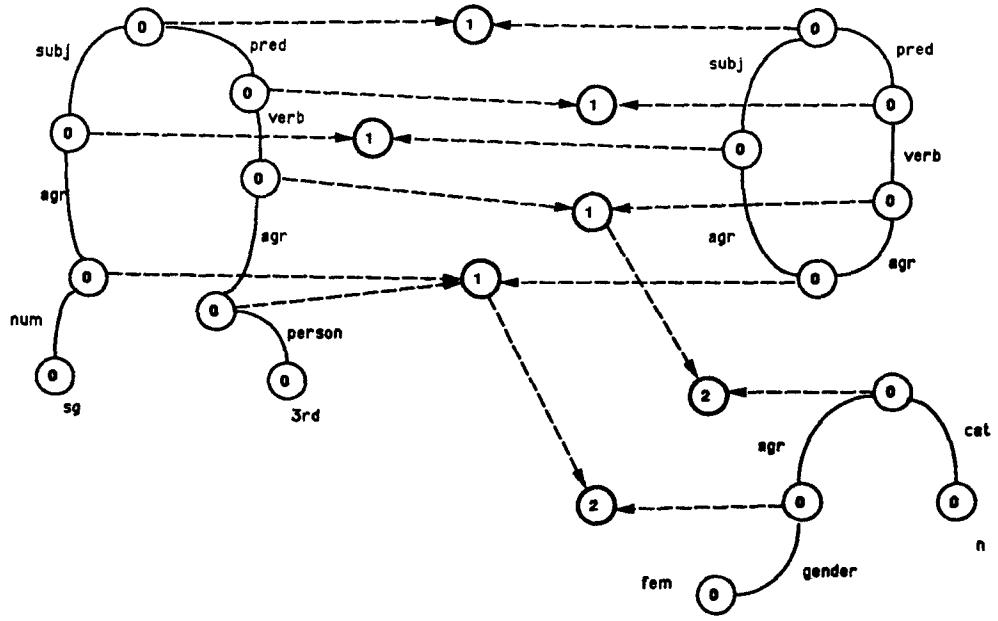

Figure 3: A unification example.

The unification algorithm is shown in Figure 4 and Figure 3 illustrates its application to a concrete example of two successive unifications. Copying the nodes that have been created by the first unification do not need to be copied again for the second unification that is applied at the node appearing as the value of the path `pred.verb`, saving five copies in comparison to the other lazy copying methods.

Another advantage of the new approach is based on the ability to switch easily between destructive and non-destructive unification. During parsing or during the process of checking the satisfiability of constraints via backtracking, there are in general several choice points. For every choice point with n possible continuations, $n-1$ lazy incremental copies of the DG are made using non-destructive unification. The last alternative continues destructively, resembling the last call optimization technique of Prolog implementations, yielding n DG representations, one for each alternative. Since each node reflects its own update history for each continuation path, all unchanged parts of the DG are shared. To sum up, derived DG instances are shared with input DG representations and updates to certain nodes by means of copy nodes are shared by different branches of the search space. Each new update corresponds to a new choice point in chronological order. The proposed environment representation facilitates memory management for allocating and deallocating copy node structures which is very important for the algorithm to be efficient. This holds, in particular, if it takes much more time to create new structures than to update old reclaimed structures.

Comparison with other Approaches

Karttunen's Reversible Unification [Karttunen 86] does not use structure sharing at all. A new DG is copied from the modified arguments after successful unification, and the argument DGs are then restored to their original state by undoing all the changes made during unification hence requiring a second pass through the DG to assemble the result and adding a constant time for the save operation before each modification.

As it has been noticed by [Godden 90] and [Kogure 90], the key idea of avoiding "redundant copying" is to do copying lazily. Copying of nodes will be delayed until a destructive change is about to take place.

Godden uses active data structures (Lisp closures) to implement lazy evaluation of copying, and Kogure uses a revised copynode procedure which maintains copy dependency information in order to avoid immediate copying.

```
procedure unify(node1,node2 : CopyNode)
  node1 ← deref(node1)
  node2 ← deref(node2)
  IF node1 = node2  THEN return(node1)
     ELSE
     newtype ← node1.type ∧ node2.type
      IF newtype = ⊥  THEN return(⊥)
         ELSE
         <SharedArcs1, SharedArcs2> ← SharedArcs(node1,node2)
         <UniqueArcs1, UniqueArcs2> ← UniqueArcs(node1,node2)
          IF ActiveP(node1)  THEN
            node ← node1
            node.arcs ←  node.arcs ∪ UniqueArcs2
            node2.copy ← node
          ELSE
          IF ActiveP(node2)  THEN
            node ← node2
            node.arcs ←  node.arcs ∪ UniqueArcs1
            node1.copy ← node
          ELSE
            node ← CreateCopyNode
            node1.copy ← node
            node2.copy ← node
            node.arcs ← UniqueArcs1 ∪ SharedArcs1 ∪ UniqueArcs2
          ENDIF
          ENDIF
            node.type ← newtype
            FOR EACH <SharedArc1, SharedArc2>
                IN <SharedArcs1, SharedArcs2>
                DO unify(SharedArc1.dest,SharedArc2.dest)
               return(node)
     ENDIF
   ENDIF
END unify
```

Figure 4: The unification procedure

approach	methods					
	early copying	over copying	redundant copying	incr. copying	lazy copying	structure sharing
naive	yes	yes	yes	no	no	no
Pereira 85	no	no	no	no	no	yes
Karttunen/Kay 85	no	no	yes	no	yes	yes
Karttunen 86	no	no	yes	no	no	no
Wroblewski 87	no	yes	yes	yes	no	no
Godden 90	no	no	yes	no	yes	yes
Kogure 90	no	yes	yes	no	yes	yes
LIC	no	yes	no	yes	yes	yes

Figure 5: Comparison of unification approaches

Both of these approaches suffer from difficulties of their own. In Godden's case, part of the copying is substituted/traded for by the creation of active data structures (Lisp closures), a potentially very costly operation, even where it would turn out that those closures remain unchanged in the final result; hence their creation is unnecessary. In addition, the search for already existing instances of active data structures in the copy environment and merging of environments for successive unifications causes an additional overhead.

Similarly, in Kogure's approach, not all redundant copying is avoided in cases where there exists a feature path (a sequence of nodes connected by arcs) to a node that needs to be copied. All the nodes along such a path must be copied, even if they are not affected by the unification procedure. Furthermore, special copy dependency information has to be maintained while copying nodes in order to trigger copying of such arc sequences leading to a node where copying is needed later in the process of unification. In addition to the overhead of storing copy dependency information, a second traversal of the set of dependent nodes is required for actually performing the copying. This copying itself might eventually trigger further copying of new dependent nodes.

The table of Figure 5 summarizes the different unification approaches that have been discussed and compares them according to the concepts and methods they use.

Conclusion

The lazy-incremental copying (LIC) method used for the unification algorithm combines incremental copying with lazy copying to achieve structure sharing. It eliminates redundant copying in all cases even where other methods still copy over. The price to be paid is counted in terms of the time spent for dereferencing but is licensed for by the gain of speed we get through the reduction both in terms of the number of copies to be made and in terms of the space required for the copies themselves.

The algorithm has been implemented in Common Lisp and runs on various workstation architectures. It is used as the essential operation in the implementation of the interpreter for the Typed Features Structure System (TFS [Emele/Zajac 90a, Emele/Zajac 90b]). The formalism of TFS is based on the notion of inheritance and sets of constraints that categories of the sort signature must satisfy. The formalism supports to express directly principles and generalizations of linguistic theories as they are formulated for example in the framework of HPSG [Pollard and Sag 87]. The efficiency of the LIC approach has been tested and compared with Wroblewski's method on a sample grammar of HPSG using a few test sentences for parsing and generation. The overall processing time is reduced by 60% - 70% of the original processing time. See [Emele 91] for further discussion of optimizations available for specialized applications of this general unification algorithm. This paper also provides a detailed metering statistics for all of the other unification algorithms that have been compared.

References

[Aït-Kaci 84] Hassan Aït-Kaci. *A Lattice Theoretic Approach to Computation based on a Calculus of Partially Ordered Types Structures.* Ph.D Dissertation, University of Pennsylvania, 1984.

[Aït-Kaci 86] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science 45*, pp. 293–351, 1986

[Boyer/Moore 72] R. S. Boyer and J. S. Moore. The sharing of structures in theorem-proving programs. In B. Meltzer and D. Mitchie (eds.), *Machine Intelligence 7*, pp. 101–116, John Wiley and Sons, New York, New York, 1972.

[Emele/Zajac 90a] Martin C. Emele and Rémi Zajac. A fix-point semantics for feature type systems. In *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems, CTRS'90*, Montréal, June 1990.

[Emele/Zajac 90b] Martin C. Emele and Rémi Zajac. Typed unification grammars. In *Proceedings of 13th International Conference on Computational Linguistics, COLING-90*, Helsinki, August 1990.

[Emele 91] Martin C. Emele. Graph Unification using Lazy Non-Redundant Copying. Technical Report AIMS 04-91, Institut für maschinelle Sprachverarbeitung, University of Stuttgart, 1991.

[Godden 90] Kurt Godden. Lazy unification. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 180–187, Pittsburgh, PA, 1990.

[Huet 76] Gérard Huet. *Résolution d'Equations dans des Langages d'Ordre 1, 2, ..., ω.* Thèse de Doctorat d'Etat, Université de Paris VII, France. 1976.

[Hopcroft/Karp 71] J. E. Hopcroft and R. M. Karp. *An Algorithm for testing the Equivalence of Finite Automata.* Technical report TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, NY, 1971.

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

[Karttunen/Kay 85] Lauri Karttunen and Martin Kay. Structure sharing with binary trees. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 133–136a, Chicago, IL, 1985.

[Kogure 90] Kiyoshi Kogure. Strategic lazy incremental copy graph unification. In *Proceedings of the 13th Intl. Conference on Computational Linguistics, COLING-90*, pp. 223–228, Helsinki, 1990.

[Pereira 85] Fernando C.N. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics, ACL*, pp. 137–144, Chicago, IL, 1985.

[Pereira/Shieber 84] Fernando C.N. Pereira and Stuart Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the 10th Intl. Conference on Computational Linguistics, COLING-84*, Stanford, 1984.

[Pollard and Sag 87] Carl Pollard and Ivan Sag. *Information-Based Syntax and Semantics, Volume I.* CSLI Lecture Notes No 13. Chicago University Press, Chicago, 1987.

[Rounds/Kasper 86] Williams C. Rounds and R. Kasper. A complete logical calculus for record structures representing linguistic information. In *IEEE Symposium on Logic in Computer Science*, 1986.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-based Approaches to Grammar.* CSLI Lecture Notes No 4. Chicago University Press, Chicago, 1986.

[Warren 83] David H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, CA, 1983.

[Wroblewski 87] David A. Wroblewski. Nondestructive graph unification. In *Proceedings of the 6th National Conference on Artificial Intelligence, AAAI*, pp. 582–587, Seattle, WA, 1987.