LAZY UNIFICATION

Kurt Godden
Computer Science Department
General Motors Research Laboratories
Warren, MI 48090-9055, USA
CSNet: godden@gmr.com

ABSTRACT

Unification-based NL parsers that copy argument graphs to prevent their destruction suffer from inefficiency. Copying is the most expensive operation in such parsers, and several methods to reduce copying have been devised with varying degrees of success. Lazy Unification is presented here as a new, conceptually elegant solution that reduces copying by nearly an order of magnitude. Lazy Unification requires no new slots in the structure of nodes, and only nominal revisions to the unification algorithm.

PROBLEM STATEMENT

Unification is widely used in natural language processing (NLP) as the primary operation during parsing. The data structures unified are directed acyclic graphs (DAG's), used to encode grammar rules, lexical entries and intermediate parsing structures. A crucial point concerning unification is that the resulting DAG is constructed directly from the raw material of its input DAG's, i.e. unification is a *destructive* operation. This is especially important when the input DAG's are rules of the grammar or lexical items. If nothing were done to prevent their destruction during unification, then the grammar would no longer have a correct rule, nor the lexicon a valid lexical entry for the DAG's in question. They would have been transformed into the unified DAG as a side effect.

The simplest way to avoid destroying grammar rules and lexical entries by unification is to copy each argument DAG prior to calling the unification routine. This is sufficient to avoid the problem of destruction, but the copying itself then becomes problematic, causing severe degradation in performance. This performance drain is illustrated in Figure 1, where average parsing statistics are given for the original implementation of graph unification in the TASLINK natural language system. TASLINK was built upon the LINK parser in a joint project between GM Research and the University of Michigan. LINK is a descendent of the MOPTRANS system developed by Lytinen (1986). The statistics below are for ten sentences parsed by TASLINK. As can be seen, copying consumes more computation time than unification.

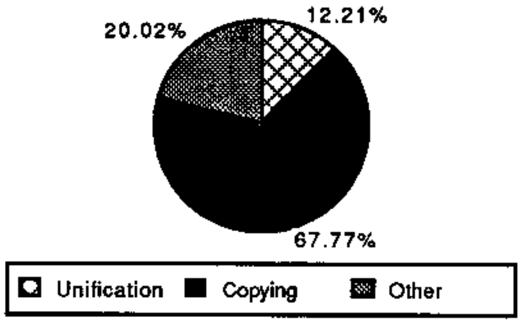


Figure 1. Relative Cost of Operations during Parsing

PAST SOLUTIONS

Improving the efficiency of unification has been an active area of research in unification-based NLP, where the focus has been on reducing the amount of DAG copying, and several approaches have arisen. Different versions of structure sharing were employed by Pereira (1985) as well as Karttunen and Kay (1985). In Karttunen (1986) structure sharing was abandoned for a technique allowing reversible unification. Wroblewski (1987) presents what he calls a non-destructive unification algorithm that avoids destruction by incrementally copying the DAG nodes as necessary.

All of these approaches to the copying problem suffer from difficulties of their own. For both Pereira and Wroblewski there are special cases involving convergent arcs—arcs from two or more nodes that point to the same destination node—that still require full copying. In Karttunen and Kay's version of structure sharing, all DAG's are represented as *binary* branching DAG's, even though grammar rules are more naturally represented as non-binary structures. Reversible unification requires two passes over the input DAG's, one to unify them and another to copy the result. Furthermore, in both successful and unsucessful unification the input DAG's must be restored to their original forms because reversible unification allows them to be destructively modified.

Wroblewski points out a useful distinction between *early* copying and *over* copying. Early copying refers to the copying of input DAG's before unification is applied. This can lead to inefficiency when unification fails because only the copying *up to* the point of failure is necessary. Over copying refers to the fact that when the two input DAG's are copied they are copied in their entirety. Since the resultant unified DAG generally has fewer total nodes than the two input DAG's, more nodes than necessary were copied to produce the result. Wroblewski's algorithm eliminates early copying entirely, but as noted above it can partially over copy on DAG's involving convergent arcs. Reversible unification may also over copy, as will be shown below.

LAZY UNIFICATION

I now present **Lazy Unification** (LU) as a new approach to the copying problem. In the following section I will present statistics which indicate that LU accomplishes *nearly an order of magnitude reduction* in copying compared to non-lazy, or *eager unification* (EU). These results are attained by turning DAG's into active data structures to implement the lazy evaluation of copying.

Lazy evaluation is an optimization technique developed for the interpretation of functional programming languages (Field and Harrison, 1988), and has been extended to theorem proving and logic programming in attempts to integrate that paradigm with functional programming (Reddy, 1986).

The concept underlying lazy evaluation is simple: delay the operation being optimized until the value it produces is needed by the calling program, at which point the delayed operation is forced. These actions may be implemented by high-level procedures called *delay* and *force*. Delay is used in place of the original call to the procedure being optimized, and force is inserted into the program at each location where the results of the delayed procedure are needed.

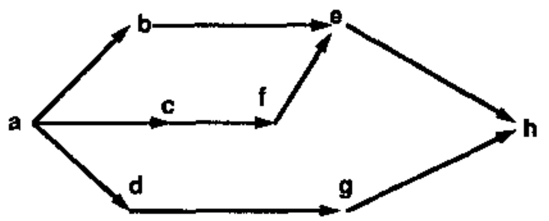
Lazy evaluation is a good technique for the copying problem in graph unification precisely because the overwhelming majority of copying is unnecessary. If all copying can be delayed until a destructive change is about to occur to a DAG, then both early copying and over copying can be completely eliminated.

The delay operation is easily implemented by using *closures*. A closure is a compound object that is both procedure and data. In the context of LU, the data portion of a closure is a DAG node. The procedural code within a closure is a function that processes a variety of messages sent to the closure. One may generally think of the encapsulated procedure as being a suspended call to the copy function. Let us refer to these closures as *active nodes* as contrasted with a *simple node* not combined with a procedure in a closure. The delay function returns an active node when given a simple node as its argument. For now let us assume that delay behaves as the identity function when applied to an active node. That is, it returns an active node unchanged. As a mnemonic we will refer to the delay function as *delay-copy-the-dag*.

We now redefine DAG's to allow either simple *or* active nodes wherever simple nodes were previously allowed in a DAG. An active node will be notated in subsequent diagrams by enclosing the node in angle brackets.

In LU the unification algorithm proceeds largely as it did before, except that at every point in the algorithm where a destructive change is about to be made to an active node, that node is first replaced by a copy of its encapsulated node. This replacement is mediated through the force function, which we shall call *force-delayed-copy*. In the case of a simple node argument force-delayed-copy acts as the identity function, but when given an active node it invokes the suspended copy procedure with the encapsulated node as argument. Force-delayed-copy returns the DAG that results from this invocation.

To avoid copying an entire DAG when only its root node is going to be modified by unification, the copying function is also rewritten. The new version of *copy-the-dag* takes an optional argument to control how much of the DAG is to be copied. The default is to copy the entire argument, as one would expect of a function called *copy-the-dag*. But when copy-the-dag is called from inside an active node (by force-delayed-copy invoking the procedural portion of the active node), then the optional argument is supplied with a flag that causes copy-the-dag to copy *only* the root node of its argument. The nodes at the ends of the outgoing arcs from the new root become active nodes, created by delaying the original nodes in those positions. No traversal of the DAG takes place and the deeper nodes are only present implicitly through the active nodes of the resulting DAG. This is illustrated in Figure 2.



becomes

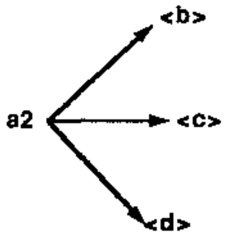


Figure 2. Copy-the-dag on 'a' from Inside an Active Node

Here, DAG **a** was initially encapsulated in a closure as an active node. When **a** is about to undergo a destructive change by being unified with some other DAG, force-delayed-copy activates the suspended call to copy-the-dag with DAG **a** as its first argument and the message **delay-arcs** as its optional argument. Copy-the-dag then copies only node **a**, returning **a$_2$** with outgoing arcs pointing at active nodes that encapsulate the original destination nodes **b**, **c**, and **d**. DAG **a$_2$** may then be unified with another DAG without destroying DAG **a**, and the unification algorithm proceeds with the active nodes <b>, <c>, and <d>. As these subdag's are modified, their nodes are likewise copied incrementally. Figure 3 illustrates this by showing DAG a$_2$ after unifying <b>. It may be seen that as active nodes are copied one by one, the resulting unified DAG is eventually constructed.

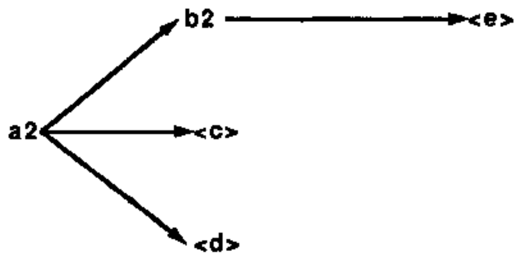


Figure 3. DAG a$_2$ after Unifying <b>

One can see how this scheme reduces the amount of copying if, for example, unification fails at the active node <e>. In this case only nodes **a** and **b** will have been copied and none of the nodes **c**, **d**, **e**, **f**, **g**, or **h**. Copying is also reduced when unification succeeds, this reduction being achieved in two ways.

First, lazy unification only creates new nodes for the DAG that *results* from unification. Generally this DAG has fewer total nodes than the two input DAG's. For example, if the 8-node DAG a in Figure 2 were unified with the 2-node DAG **a—>i**, then the resulting DAG would have only nine nodes, not ten. The result DAG would have the arc '**—>i**' copied onto the 8-node DAG's root. Thus, while EU would copy all ten original nodes, only nine are necessary for the result.

Active nodes that remain in a final DAG represent the other savings for successful unification. Whereas EU copies all ten original nodes to create the 9-node result, LU would only create five new nodes during unification, resulting in the DAG of Figure 4. Note that the "missing" nodes **e, f, g,** and **h** are implicit in the active nodes and did not require copying. For larger DAG's, this kind of savings in node copying can be significant as several large sub-DAG's may survive uncopied in the final DAG .

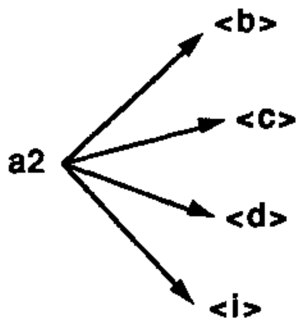


Figure 4. Saving Four Node Copies with Active Nodes

A useful comparison with Karttunen's reversible unification may now be made. Recall that when reversible unification is successful the resulting DAG is copied and the originals restored. Notice that this copying of the entire resulting DAG may *overcopy* some of the sub-DAG's. This is evident because we have just seen in LU that some of the sub-DAG's of a resulting DAG remain uncopied inside active nodes. Thus, LU offers less real copying than reversible unification.

Let us look again at DAG **a** in Figure 2 and discuss a potential problem with lazy unification as described thus far. Let us suppose that through unification **a** has been partially copied resulting in the DAG shown in Figure 5, with active node <f> about to be copied.

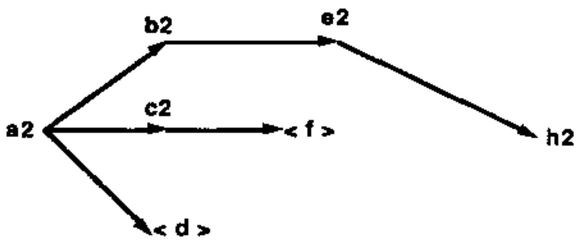


Figure 5. DAG 'a' Partially Copied

Recall from Figure 2 that node **f** points at **e**. Following the procedure described above, <f> would be copied to f_2 which would then point at active node <e>, which could lead to *another* node e_3 as shown in Figure 6. What is needed is some form of *memory* to recognize that **e** was already copied once and that f_2 needs to point at e_2 not <e>.

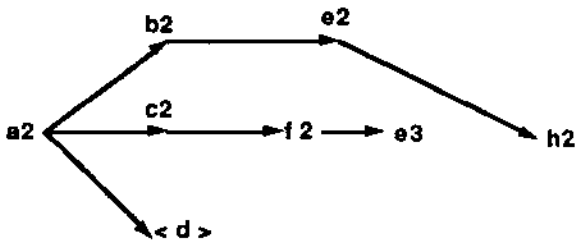


Figure 6. Erroneous Splitting of Node e into e_2 and e_3

This memory is implemented with a *copy environment*, which is an association list relating original nodes to their copies. Before f_2 is given an arc pointing at <e>, this alist is searched to see if **e** has already been copied. Since it has, e_2 is returned as the destination node for the outgoing arc from f_2, thus preserving the topography of the original DAG.

Because there are several DAG's that must be preserved during the course of parsing, the copy environment cannot be global but must be associated with each DAG for which it records the copying history. This is accomplished by encapsulating a particular DAG's copy environment in each of the active nodes of that DAG. Looking again at Figure 2, the active nodes for DAG **a_2** are all created in the scope of a variable bound to an initially empty association list for **a_2**'s copy environment. Thus, the closures that implement the active nodes <b>, <c>, and <d> all have access to the *same* copy environment. When **<b>** invokes the suspended call to *copy-the-dag*, this function adds the pair **(b . b_2)** to the copy environment as a side effect before returning its value **b_2**. When this occurs, **<c>** and **<d>** instantly have access to the new pair through their shared access to the same copy environment. Furthermore, when new active nodes are created as traversal of the DAG continues during unification, they are also created in the scope of the same copy environment. Thus, this alist is *pushed forward* deeper into the nodes of the parent DAG as part of the data portion of each active node.

Returning to Figure 5, the pair **(e . e_2)** was added to the copy environment being maintained for DAG **a_2** when **e** was copied to **e_2**. Active node <f> was created in the scope of this list and therefore "remembers" at the time **f_2** is created that it should point to the previously created **e_2** and not to a new active node <e>.

There is one more mechanism needed to correctly implement copy environments. We have already seen how some active nodes remain after unification. As intermediate DAG's are reused during the nondeterministic parsing and are unified with other DAG's, it can happen that some of these remaining active nodes become descendents of a root different from their original root node. As those new root DAG's are incrementally copied during unification, a situation can arise whereby an active node's parent node is copied and then an attempt is made to create an active node out of an active node.

For example, let us suppose that the DAG shown in Figure 5 is a sub-DAG of some larger DAG. Let us refer to the root of that larger DAG as node **n**. As unification of **n** proceeds, we may reach **a_2** and start incrementally copying it. This could eventually result in **c_2** being copied to **c_3** at which point the system will attempt to create an outgoing arc for **c_3** pointing at a newly created active node over the already active node **<f>**. There is no need to try to create such a beast as <<f>>. Rather, what is needed is to assure that active node **<f>** be given access to the *new* copy environment for **n** passed down to **<f>** from its predecessor nodes. This is accomplished by *destructively merging* the new copy environment with that previously created for **a_2** and surviving inside **<f>**. It is important that this merge be destructive in order to give all active nodes that are descendents of **n** access to the same information so that the problem of node splitting illustrated in Figure 6 continues to be avoided.

It was mentioned previously how calls to *force-delayed-copy* must be inserted into the unification algorithm to invoke the incremental copying of nodes. Another modification to the algorithm is also necessary as a result of this incremental copying. Since active nodes are *replaced* by new nodes in the middle of unification, the algorithm must undergo a revision to effect this replacement. For example, in Figure 5 in order for **<b>** to be replaced by **b_2**, the corresponding arc from **a_2** must be replaced. Thus as the unification algorithm traverses a DAG, it also collects such replacements in order to reconstruct the outgoing arcs of a parent DAG.

In addition to the message **delay-arcs** sent to an active node to invoke the suspended call to copy-the-dag, other messages are needed. In order to compare active nodes and merge their copy environments, the active nodes must process messages that cause the active node to return either its encapsulated node's label or the encapsulated copy environment.

EFFECTIVENESS OF LAZY UNIFICATION

Lazy Unification results in an impressive reduction to the amount of copying during parsing. This in turn reduces the overall slice of parse time consumed by copying as can be seen by contrasting Figure 7 with Figure 1. Keep in mind that these charts illustrate *proportional* computations, not speed. The pie shown below should be viewed as a smaller pie, representing faster parse times, than that in Figure 1. Speed is discussed below.

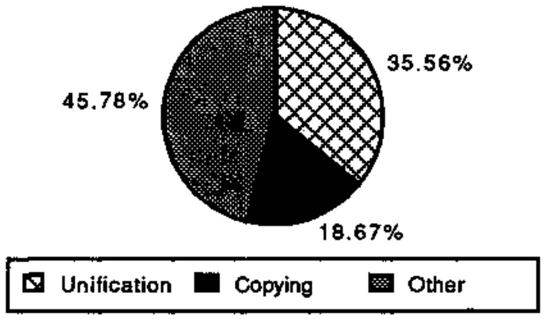


Figure 7. Relative Cost of Operations with Lazy Unification

Lazy Unification copies less than 7% of the nodes copied under eager unification. However, this is not a fair comparison with EU because LU substitutes the creation of active nodes for some of the copying. To get a truer comparison of Lazy vs. Eager Unification, we must add together the number of copied nodes and active nodes created in LU. Even when active nodes are taken into account, the results are highly favorable toward LU because again less than 7% of the nodes copied under EU are accounted for by active nodes in LU. Combining the active nodes with copies, LU still accounts for an 87% reduction over eager unification. Figure 8 graphically illustrates this difference for ten sentences.

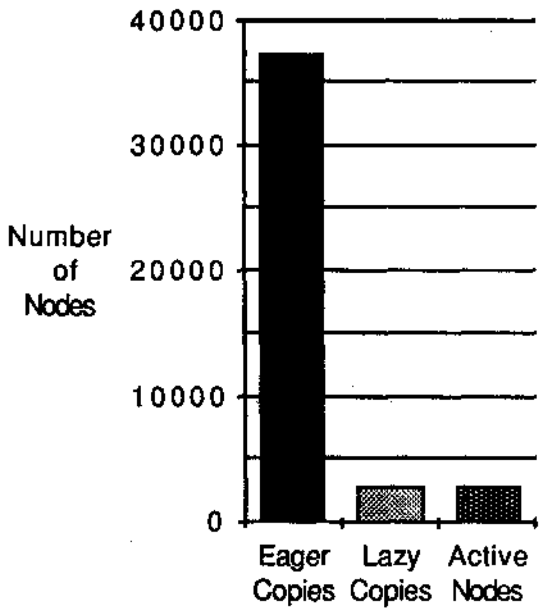


Figure 8. Comparison of Eager vs. Lazy Unification

From the time slice of eager copying shown in Figure 1, we can see that if LU were to incur no overhead then an 87% reduction of copying would result in a faster parse of roughly 59%. The actual speedup is about 50%, indicating that the overhead of implementing LU is 9%. However, the 50% speedup does not consider the effects of garbage collection or paging since they are system dependent. These effects will be more pronounced in EU than LU because in the former paradigm more data structures are created and referenced. In practice, therefore, LU performs at better than twice the speed of EU.

There are several sources of overhead in LU. The major cost is incurred in distinguishing between active and simple nodes. In our Common Lisp implementation simple DAG nodes are defined as named *structures* and active nodes as *closures*. Hence, they are distinguished by the Lisp predicates **DAG-P** and **FUNCTIONP**. Disassembly on a Symbolics machine shows both predicates to be rather costly. (The functions **TYPE-OF** and **TYPEP** could also be used, but they are also expensive.)

Another expensive operation occurs when the copy environments in active nodes are searched. Currently, these environments are simple association lists which require sequential searching. As was discussed above, the copy environments must sometimes be merged. The merge function presently uses the **UNION** function. While a far less expensive destructive concatenation of copy environments could be employed, the union operation was chosen initially as a simple way to avoid creation of circular lists during merging.

All of these sources of overhead can and will be attacked by additional work. Nodes can be defined as a tagged data structure, allowing an inexpensive tag test to distinguish between active and inactive nodes. A non-sequential data structure could allow faster than linear searching of copy environments and more efficient merging. These and additional modifications are expected to eliminate most of the overhead incurred by the current implementation of LU. In any case, Lazy Unification was developed to reduce the amount of copying during unification and we have seen its dramatic success in achieving that goal.

CONCLUDING REMARKS

There is another optimization possible regarding certain leaf nodes of a DAG. Depending on the application using graph unification, a subset of the leaf nodes will never be unified with other DAG's. In the TASLINK application these are nodes representing such features as third person singular. This observation can be exploited under both lazy and eager unification to reduce both copying and active node creation. See Godden (1989) for details.

It has been my experience that using lazy evaluation as an optimization technique for graph unification, while elegant in the end result, is slow in development time due to the difficulties it presents for debugging. This property is intrinsic to lazy evaluation, (O'Donnell and Hall, 1988).

The problem is that a DAG is no longer copied locally because the copy operation is suspended in the active nodes. When a DAG *is* eventually copied, that copying is performed incrementally and therefore non-locally in both time and program space. In spite of this distributed nature of the optimized process, the programmer continues to conceptualize the operation as occurring locally as it would occur in the non-optimized eager mode. As a result of this mismatch between the programmer's visualization of the operation and its actual execution, bugs are notoriously difficult to trace. The development time for a program employing lazy evaluation is, therefore, much longer than would be expected. Hence, this technique should only be employed when the possible efficiency gains are expected to be large, as they are in the case of graph unification. O'Donnell and Hall present an excellent discussion of these and other problems and offer insight into how tools may be built to alleviate some of them.

REFERENCES

Field, Anthony J. and Peter G. Harrison. 1988. *Functional Programming*. Reading, MA: Addison-Wesley.

Godden, Kurt. 1989. "Improving the Efficiency of Graph Unification." Internal technical report GMR-6928. General Motors Research Laboratories. Warren, MI.

Karttunen, Lauri. 1986. *D-PATR: A Development Environment for Unification-Based Grammars*. Report No. CSLI-86-61. Stanford, CA.

Karttunen, Lauri and Martin Kay. 1985. "Structure-Sharing with Binary Trees." *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*. Chicago, IL: ACL. pp. 133-136A.

Lytinen, Steven L. 1986. "Dynamically Combining Syntax and Semantics in Natural Language Processing." *Proceedings of the*

5th National Conference on Artificial Intelligence. Philadelphia, PA: AAAI. pp. 574–578.

O'Donnell, John T. and Cordelia V. Hall. 1988. "Debugging in Applicative Languages." *Lisp and Symbolic Computation,* 1/2. pp. 113–145.

Pereira, Fernando C. N. 1985. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms." *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.* Chicago, IL: ACL. pp. 137–144.

Reddy, Uday S. 1986. "On the Relationship between Logic and Functional Languages," in Doug DeGroot and Gary Lindstrom, eds. *Logic Programming: Functions, Relations, and Equations.* Englewood Cliffs, NJ. Prentice-Hall. pp. 3–36.

Wroblewski, David A. 1987. "Nondestructive Graph Unification." *Proceedings of the 6th National Conference on Artificial Intelligence.* Seattle, WA: AAAI. pp. 582–587.