Memory-Efficient and Thread-Safe Quasi-Destructive Graph Unification

Marcel P. van Lohuizen
Department of Information Technology and Systems
Delft University of Technology
mpvl@acm.org

Abstract

In terms of both speed and memory consumption, graph unification remains the most expensive component of unification-based grammar parsing. We present a technique to reduce the memory usage of unification algorithms considerably, without increasing execution times. Also, the proposed algorithm is thread-safe, providing an efficient algorithm for parallel processing as well.

1 Introduction

Both in terms of speed and memory consumption, graph unification remains the most expensive component in unification-based grammar parsing. Unification is a well known algorithm. Prolog, for example, makes extensive use of term unification. Graph unification is slightly different. Two different graph notations and an example unification are shown in Figure 1 and 2, respectively.

In typical unification-based grammar parsers, roughly 90% of the unifications fail. Any processing to create, or copy, the result graph before the point of failure is redundant. As copying is the most expensive part of unification, a great deal of research has gone in eliminating superfluous copying. Examples of these approaches are given in (Tomabechi, 1991) and (Wroblewski, 1987). In order to avoid superfluous copying, these algorithms incorporate control data in the graphs. This has several drawbacks, as we will discuss next.

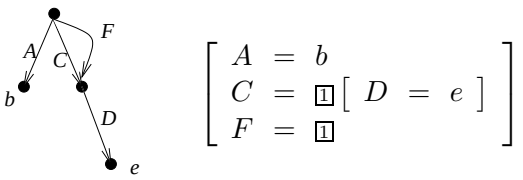

Figure 1: Two ways to represent an identical graph.

Memory Consumption To achieve the goal of eliminating superfluous copying, the aforementioned algorithms include administrative fields—which we will call **scratch fields**—in the node structure. These fields do not attribute to the definition of the graph, but are used to efficiently guide the unification and copying process. Before a graph is used in unification, or after a result graph has been copied, these fields just take up space. This is undesirable, because memory usage is of great concern in many unification-based grammar parsers. This problem is especially of concern in Tomabechi's algorithm, as it increases the node size by at least 60% for typical implementations.

In the ideal case, scratch fields would be stored in a separate buffer allowing them to be reused for each unification. The size of such a buffer would be proportional to the maximum number of nodes that are involved in a single unification. Although this technique reduces memory usage considerably, it does not reduce the amount of data involved in a single unification. Nevertheless, storing and loading nodes without scratch fields will be faster, because they are smaller. Because scratch fields are reused, there is a high probability that they will remain in cache. As the difference

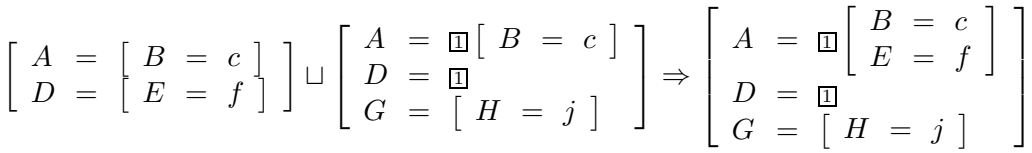

Figure 2: An example unification in attribute value matrix notation.

in speed between processor and memory continues to grow, caching is an important consideration (Ghosh et al., 1997).[1]

A straightforward approach to separate the scratch fields from the nodes would be to use a hash table to associate scratch structures with the addresses of nodes. The overhead of a hash table, however, may be significant. In general, any binding mechanism is bound to require some extra work. Nevertheless, considering the difference in speed between processors and memory, reducing the memory footprint may compensate for the loss of performance to some extent.

Symmetric Multi Processing Small-scale desktop multiprocessor systems (e.g. dual or even quad Pentium machines) are becoming more commonplace and affordable. If we focus on graph unification, there are two ways to exploit their capabilities. First, it is possible to parallelize a single graph unification, as proposed by e.g. (Tomabechi, 1991). Suppose we are unifying graph a with graph b, then we could allow multiple processors to work on the unification of a and b simultaneously. We will call this **parallel unification**. Another approach is to allow multiple graph unifications to run concurrently. Suppose we are unifying graph a and b in addition to unifying graph a and c. By assigning a different processor to each operation we obtain what we will call **concurrent unification**. Parallel unification exploits parallelism inherent of graph unification itself, whereas concurrent unification exploits parallelism at the context-free grammar backbone. As long as the number of unification operations in one parse is large, we believe it is preferable to choose concurrent unification. Especially when a large number of unifications terminates quickly (e.g. due to failure), the overhead of more finely grained parallelism can be considerable.

In the example of concurrent unification, graph a was used in both unifications. This suggests that in order for concurrent unification to work, the input graphs need to be read only. With destructive unification algorithms this does not pose a problem, as the source graphs are copied before unification. However, including scratch fields in the node structure (as Tomabechi's and Wroblewski's algorithms do) thwarts the implementation of concurrent unification, as different processors will need to write different values in these fields. One way to solve this problem is to disallow a single graph to be used in multiple unification operations simultaneously. In (van Lohuizen, 2000) it is shown, however, that this will greatly impair the ability to achieve speedup. Another solution is to duplicate the scratch fields in the nodes for each processor. This, however, will enlarge the node size even further. In other words, Tomabechi's and Wroblewski's algorithms are not suited for concurrent unification.

2 Algorithm

The key to the solution of all of the abovementioned issues is to separate the scratch fields from the fields that actually make up the definition of the graph. The resulting data structures are shown in Figure 3. We have taken Tomabechi's quasi-destructive graph unification algorithm as the starting point (Tomabechi, 1995), because it is often considered to be the fastest unification algo-

[1] Most of today's computers load and store data in large chunks (called cache lines), causing even uninitialized fields to be transported.

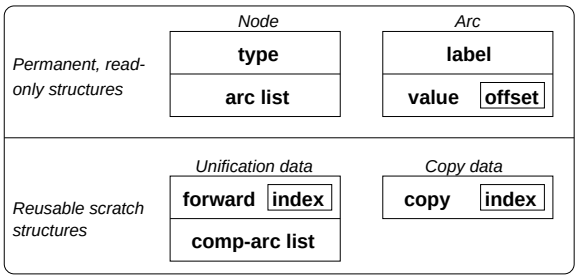

Figure 3: Node and Arc structures and the reusable scratch fields. In the permanent structures we use offsets. Scratch structures use index values (including arcs recorded in comp-arc list). Our implementation derives offsets from index values stored in nodes.

rithm for unification-based grammar parsing (see e.g. (op den Akker et al., 1995)). We have separated the scratch fields needed for unification from the scratch fields needed for copying.[2]

We propose the following technique to associate scratch structures with nodes. We take an array of scratch structures. In addition, for each graph we assign each node a unique index number that corresponds to an element in the array. Different graphs typically share the same indexes. Since unification involves two graphs, we need to ensure that two nodes will not be assigned the same scratch structure. We solve this by interleaving the index positions of the two graphs. This mapping is shown in Figure 4. Obviously, the minimum number of elements in the table is two times the number of nodes of the largest graph. To reduce the table size, we allow certain nodes to be deprived of scratch structures. (For example, we do not forward atoms.) We denote this with a valuation function v, which returns 1 if the node is assigned an index and 0 otherwise.

We can associate the index with a node by including it in the node structure. For structure sharing, however, we have to use offsets between nodes (see Figure 4), because otherwise different nodes in a graph may end up having the same index (see Section 3). Offsets can be easily derived from index values in nodes. As storing offsets in arcs consumes more memory than storing indexes in nodes (more arcs may point to the same node), we store index values and use them to compute the offsets. For ease of reading, we present our algorithm as if the offsets were stored instead of computed. Note that the small index values consume much less space than the scratch fields they replace.

[2]The arc-list field could be used for permanent forward links, if required.

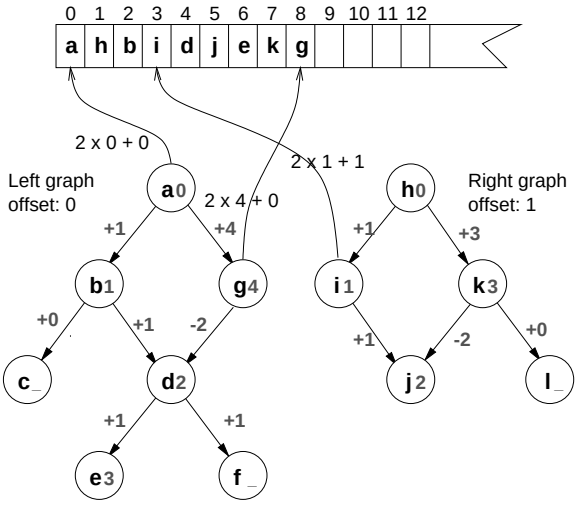

Figure 4: The mechanism to associate index numbers with nodes. The numbers in the nodes represent the index number. Arcs are associated with offsets. Negative offsets indicate a reentrancy.

The resulting algorithm is shown in Figure 5. It is very similar to the algorithm in (Tomabechi, 1991), but incorporates our indexing technique. Each reference to a node now not only consists of the address of the node structure, but also its index in the table. This is required because we cannot derive its table index from its node structure alone.

The second argument of COPY indicates the next free index number. COPY returns references with an offset, allowing them to be directly stored in arcs. These offsets will be negative when COPY exits at line 2.2, resembling a reentrancy. Note that only ABSARC explicitly defines operations on offsets. ABSARC computes a node's index using its parent node's index and an offset.

```
UNIFY(dg1, dg2)
 1. try UNIFY1((dg1, 0), (dg2, 1))[a]
      1.1. (copy, n) ← COPY((dg1, 0), 0)
      1.2. Clear the fwtab and cptab table.[b]
      1.3. return copy
 2. catch
      2.1. Clear the fwtab table.[b]
      2.2. return nil

UNIFY1(ref_in1, ref_in2)
 1. ref1 ← (dg1, idx1) ← DEREFERENCE(ref_in1)
 2. ref2 ← (dg2, idx2) ← DEREFERENCE(ref_in2)
 3. if dg1 ≡addr dg2 and idx1 = idx2[c] then
      3.1. return
 4. if dg1.type = bottom then
      4.1. FORWARD(ref1, ref2)
 5. elseif dg2.type = bottom then
      5.1. FORWARD(ref2, ref1)
 6. elseif both dg1 and dg2 are atomic then
      6.1. if dg1.arcs ≠ dg2.arcs then
             throw UnificationFailedException
      6.2. FORWARD(ref2, ref1)
 7. elseif either dg1 or dg2 is atomic then
      7.1. throw UnificationFailedException
 8. else
      8.1. FORWARD(ref2, ref1)
      8.2. shared ← INTERSECTARCS(ref1, ref2)
      8.3. for each ((_, r1), (_, r2)) in shared do
             UNIFY1(r1, r2)
      8.4. new ← COMPLEMENTARCS(ref1, ref2)
      8.5. for each arc in new do
             Push arc to fwtab[idx1].comp_arcs

FORWARD((dg1, idx1), (dg2, idx2))
 1. if v(dg1) = 1 then
      fwtab[idx1].forward ← (dg2, idx2)

ABSARC((label, (dg, off)), current_idx)
   return (label, (dg, current_idx + 2 · off))[d]

DEREFERENCE((dg, idx))
 1. if v(dg1) = 1 then
      1.1. (fwd-dg, fwd-idx) ← fwtab[idx].forward
      1.2. if fwd-dg ≠ nil then
             DEREFERENCE(fwd-dg, fwd-idx)
      1.3. else
             return (dg, idx)

INTERSECTARCS(ref1, ref2)
   Returns pairs of arcs with index values for each pair
   of arcs in ref1 resp. ref2 that have the same label.
   To obtain index values, arcs from arc-list must be
   converted with ABSARC.

COMPLEMENTARCS(ref1, ref2)
   Returns node references for all arcs with labels that
   exist in ref2, but not in ref1. The references are com-
   puted as with INTERSECTARCS.

COPY(ref_in, new_idx)
 1. (dg, idx) ← DEREFERENCE(ref_in)
 2. if v(dg) = 1 and cptab[idx].copy ≠ nil then
      2.1. (dg1, idx1) ← cptab[idx].copy
      2.2. return (dg1, idx1 − new_idx + 1)
 3. newcopy ← new Node
 4. newcopy.type ← dg.type
 5. if v(dg) = 1 then
      cptab[idx].copy ← (newcopy, new_idx)
 6. count ← v(newcopy)[e]
 7. if dg.type = atomic then
      7.1. newcopy.arcs ← dg.arcs
 8. elseif dg.type = complex then
      8.1. arcs ← {ABSARC(a, idx) | a ∈ dg.arcs}
      ∪ fwtab[idx].comp_arcs
      8.2. for each (label, ref) in arcs do
             ref1 ← COPY(ref, count + new_idx)[f]
             Push (label, ref1) into newcopy.arcs
             if ref1.offset > 0[g] then
               count ← count + ref1.offset
 9. return (newcopy, count)
```

[a] We assign even and odd indexes to the nodes of dg1 and dg2, respectively.
[b] Tables only needs to be cleared up to point where unification failed.
[c] Compare indexes to allow more powerful structure sharing. Note that indexes uniquely identify a node in the case that for all nodes n holds $v(n) = 1$.
[d] Note that we are multiplying the offset by 2 to account for the interleaved offsets of the left and right graph.
[e] We assume it is known at this point whether the new node requires an index number.
[f] Note that ref contains an index, whereas ref1 contains an offset.
[g] If the node was already copied (in which case it is < 0), we need not reserve indexes.

Figure 5: The memory-efficient and thread-safe unification algorithm. Note that the arrays fwtab and cptab—which represent the forward table and copy table, respectively—are defined as global variables. In order to be thread safe, each thread needs to have its own copy of these tables.

Contrary to Tomabechi's implementation, we invalidate scratch fields by simply resetting them after a unification completes. This simplifies the algorithm. We only reset the table up to the highest index in use. As table entries are roughly filled in increasing order, there is little overhead for clearing unused elements.

A nice property of the algorithm is that indexes identify from which input graph a node originates (even=left, odd=right). This information can be used, for example, to selectively share nodes in a structure sharing scheme. We can also specify additional scratch fields or additional arrays at hardly any cost. Some of these abilities will be used in the enhancements of the algorithm we will discuss next.

3 Enhancements

Structure Sharing Structure sharing is an important technique to reduce memory usage. We will adopt the same terminology as Tomabechi in (Tomabechi, 1992). That is, we will use the term feature-structure sharing when two arcs in one graph converge to the same node in that graph (also refered to as reentrancy) and data-structure sharing when arcs from two different graphs converge to the same node.

The conditions for sharing mentioned in (Tomabechi, 1992) are: (1) bottom and atomic nodes can be shared; (2) complex nodes can be shared unless they are modified. We need to add the following condition: (3) all arcs in the shared subgraph must have the same offsets as the subgraph that would have resulted from copying. A possible violation of this constraint is shown in Figure 6. As long as arcs are processed in increasing order of index number,[3] this condition can only be violated in case of reentrancy. Basically, the condition can be violated when a reentrancy points past a node that is bound to a larger subgraph.

[3]This can easily be accomplished by fixing the order in which arcs are stored in memory. This is a good idea anyway, as it can speedup the ComplementArcs and IntersectArcs operations.

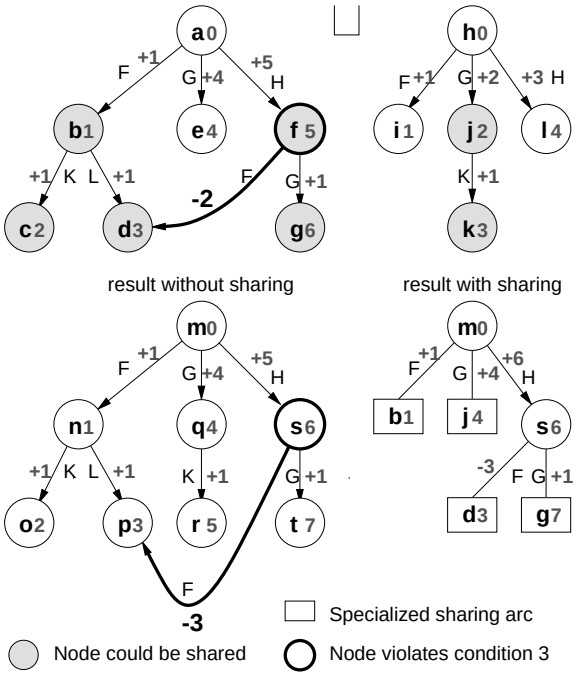

Figure 6: Sharing mechanism. Node **f** cannot be shared, as this would cause the arc labeled F to derive an index colliding with node **q**.

Contrary to many other structure sharing schemes (like (Malouf et al., 2000)), our algorithm allows sharing of nodes that are part of the grammar. As nodes from the different input graphs are never assigned the same table entry, they are always bound independently of each other. (See the footnote for line 3 of Unify1.)

The sharing version of Copy is similar to the variant in (Tomabechi, 1992). The extra check can be implemented straightforwardly by comparing the old offset with the offset for the new nodes. Because we derive the offsets from index values associated with nodes, we need to compensate for a difference between the index of the shared node and the index it should have in the new graph. We store this information in a specialized **share arc**. We need to adjust Unify1 to handle share arcs accordingly.

Deferred Copying Just as we use a table for unification and copying, we also use a table for subsumption checking. Tomabechi's algorithm requires that the graph resulting

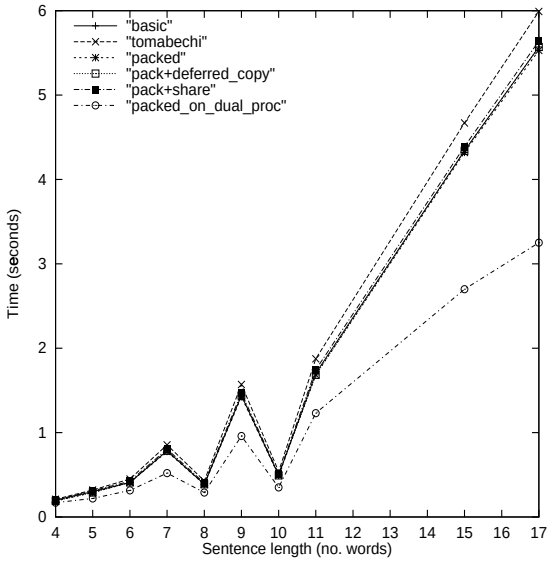

Figure 7: Execution time (seconds).

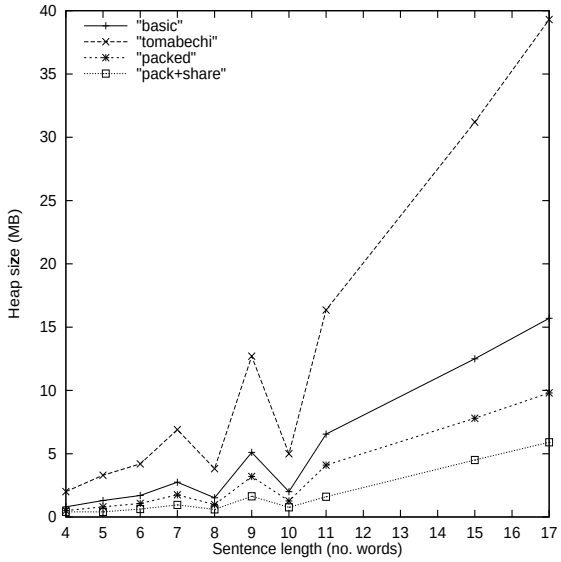

Figure 8: Memory used by graph heap (MB).

from unification be copied before it can be used for further processing. This can result in superfluous copying when the graph is subsumed by an existing graph. Our technique allows subsumption to use the bindings generated by UNIFY1 in addition to its own table. This allows us to defer copying until we completed subsumption checking.

Packed Nodes With a straightforward implementation of our algorithm, we obtain a node size of 8 bytes.[4] By dropping the concept of a fixed node size, we can reduce the size of atom and bottom nodes to 4 bytes. Type information can be stored in two bits. We use the two least significant bits of pointers (which otherwise are 0) to store this type information. Instead of using a pointer for the value field, we store nodes in place. Only for reentrancies we still need pointers. Complex nodes require 8 bytes, as they include a pointer to the first node past its children (necessary for unification). This scheme requires some extra logic to decode nodes, but significantly reduces memory consumption.

[4]We do not have a type hierarchy.

4 Experiments

We have tested our algorithm with a medium-sized grammar for Dutch. The system was implemented in Objective-C using a fixed arity graph representation. We used a test set of 22 sentences of varying length. Usually, approximately 90% of the unifications fails. On average, graphs consist of 60 nodes. The experiments were run on a Pentium III 600EB (256 KB L2 cache) box, with 128 MB memory, running Linux.

We tested both memory usage and execution time for various configurations. The results are shown in Figure 7 and 8. It includes a version of Tomabechi's algorithm. The node size for this implementation is 20 bytes. For the proposed algorithm we have included several versions: a basic implementation, a packed version, a version with deferred copying, and a version with structure sharing. The basic implementation has a node size of 8 bytes, the others have a variable node size. Whenever applicable, we applied the same optimizations to all algorithms. We also tested the speedup on a dual Pentium II 266 Mhz.[5] Each processor was assigned its own scratch tables. Apart from that, no changes to the

[5]These results are scaled to reflect the speedup relative to the tests run on the other machine.

algorithm were required. For more details on the multi-processor implementation, see (van Lohuizen, 1999).

The memory utilization results show significant improvements for our approach.[6] Packing decreased memory utilization by almost 40%. Structure sharing roughly halved this once more.[7] The third condition prohibited sharing in less than 2% of the cases where it would be possible in Tomabechi's approach.

Figure 7 shows that our algorithm does not increase execution times. Our algorithm even scrapes off roughly 7% of the total parsing time. This speedup can be attributed to improved cache utilization. We verified this by running the same tests with cache disabled. This made our algorithm actually run slower than Tomabechi's algorithm. Deferred copying did not improve performance. The additional overhead of dereferencing during subsumption was not compensated by the savings on copying. Structure sharing did not significantly alter the performance as well. Although, this version uses less memory, it has to perform additional work.

Running the same tests on machines with less memory showed a clear performance advantage for the algorithms using less memory, because paging could be avoided.

5 Related Work

We reduce memory consumption of graph unification as presented in (Tomabechi, 1991) (or (Wroblewski, 1987)) by separating scratch fields from node structures. Pereira's (Pereira, 1985) algorithm also stores changes to nodes separate from the graph. However, Pereira's mechanism incurs a $\log(n)$ overhead for accessing the changes (where n is the number of nodes in a graph), resulting in an $O(n \log n)$ time algorithm. Our algorithm runs in $O(n)$ time.

[6]The results do not include the space consumed by the scratch tables. However, these tables do not consume more than 10 KB in total, and hence have no significant impact on the results.

[7]Because the packed version has a variable node size, structure sharing yielded less relative improvements than when applied to the basic version. In terms of number of nodes, though, the two results were identical.

With respect to over and early copying (as defined in (Tomabechi, 1991)), our algorithm has the same characteristics as Tomabechi's algorithm. In addition, our algorithm allows to postpone the copying of graphs until after subsumption checks complete. This would require additional fields in the node structure for Tomabechi's algorithm.

Our algorithm allows sharing of grammar nodes, which is usually impossible in other implementations (Malouf et al., 2000). A weak point of our structure sharing scheme is its extra condition. However, our experiments showed that this condition can have a minor impact on the amount of sharing.

We showed that compressing node structures allowed us to reduce memory consumption by another 40% without sacrificing performance. Applying the same technique to Tomabechi's algorithm would yield smaller relative improvements (max. 20%), because the scratch fields cannot be compressed to the same extent.

One of the design goals of Tomabechi's algorithm was to come to an efficient implementation of parallel unification (Tomabechi, 1991). Although theoretically parallel unification is hard (Vitter and Simons, 1986), Tomabechi's algorithm provides an elegant solution to achieve limited scale parallelism (Fujioka et al., 1990). Since our algorithm is based on the same principles, it allows parallel unification as well. Tomabechi's algorithm, however, is not thread-safe, and hence cannot be used for concurrent unification.

6 Conclusions

We have presented a technique to reduce memory usage by separating scratch fields from nodes. We showed that compressing node structures can further reduce the memory footprint. Although these techniques require extra computation, the algorithms still run faster. The main reason for this was the difference between cache and memory speed. As current developments indicate that this difference will only get larger, this effect is not just an artifact of the current architectures.

We showed how to incoporate data-

structure sharing. For our grammar, the additional constraint for sharing did not pose a problem. If it does pose a problem, there are several techniques to mitigate its effect. For example, one could reserve additional indexes at critical positions in a subgraph (e.g. based on type information). These can then be assigned to nodes in later unifications without introducing conflicts elsewhere. Another technique is to include a tiny table with repair information in each share arc to allow a small number of conflicts to be resolved.

For certain grammars, data-structure sharing can also significantly reduce execution times, because the equality check (see line 3 of UNIFY1) can intercept shared nodes with the same address more frequently. We did not exploit this benefit, but rather included an offset check to allow grammar nodes to be shared as well. One could still choose, however, not to share grammar nodes.

Finally, we introduced deferred copying. Although this technique did not improve performance, we suspect that it might be beneficial for systems that use more expensive memory allocation and deallocation models (like garbage collection).

Since memory consumption is a major concern with many of the current unification-based grammar parsers, our approach provides a fast and memory-efficient alternative to Tomabechi's algorithm. In addition, we showed that our algorithm is well suited for concurrent unification, allowing to reduce execution times as well.

References

[Fujioka et al.1990] T. Fujioka, H. Tomabechi, O. Furuse, and H. Iida. 1990. Parallelization technique for quasi-destructive graph unification algorithm. In *Information Processing Society of Japan SIG Notes 90-NL-80*.

[Ghosh et al.1997] S. Ghosh, M. Martonosi, and S. Malik. 1997. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 317–324, New York, July 7–11. ACM Press.

[Malouf et al.2000] Robert Malouf, John Carroll, and Ann Copestake. 2000. Efficient feature structure operations witout compilation. *Natural Language Engineering*, 1(1):1–18.

[op den Akker et al.1995] R. op den Akker, H. ter Doest, M. Moll, and A. Nijholt. 1995. Parsing in dialogue systems using typed feature structures. Technical Report 95-25, Dept. of Computer Science, University of Twente, Enschede, The Netherlands, September. Extended version of an article published in E...

[Pereira1985] Fernando C. N. Pereira. 1985. A structure-sharing representation for unification-based grammar formalisms. In *Proc. of the 23rd Annual Meeting of the Association for Computational Linguistics. Chicago, IL, 8–12 Jul 1985*, pages 137–144.

[Tomabechi1991] H. Tomabechi. 1991. Quasi-destructive graph unifications. In *Proceedings of the 29th Annual Meeting of the ACL*, Berkeley, CA.

[Tomabechi1992] Hideto Tomabechi. 1992. Quasi-destructive graph unifications with structure-sharing. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING-92)*, Nantes, France.

[Tomabechi1995] Hideto Tomabechi. 1995. Design of efficient unification for natural language. *Journal of Natural Language Processing*, 2(2):23–58.

[van Lohuizen1999] Marcel van Lohuizen. 1999. Parallel processing of natural language parsers. In *PARCO '99*. Paper accepted (8 pages), to appear soon.

[van Lohuizen2000] Marcel P. van Lohuizen. 2000. Exploiting parallelism in unification-based parsing. In *Proc. of the Sixth International Workshop on Parsing Technologies (IWPT 2000)*, Trento, Italy.

[Vitter and Simons1986] Jeffrey Scott Vitter and Roger A. Simons. 1986. New classes for parallel complexity: A study of unification and other complete problems for $\mathcal{P}$. *IEEE Transactions on Computers*, C-35(5):403–418, May.

[Wroblewski1987] David A. Wroblewski. 1987. Nondestructive graph unification. In Howard Forbus, Kenneth; Shrobe, editor, *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*, pages 582–589, Seattle, WA, July. Morgan Kaufmann.