Natural Language Engineering **1** (1): 1–18. Printed in the United Kingdom

Efficient feature structure operations without compilation

ROBERT MALOUF
Alfa Informatica, University of Groningen,
Postbus 716, 9700 AS Groningen, The Netherlands
e-mail: `malouf@let.rug.nl`

JOHN CARROLL
Cognitive and Computing Sciences, University of Sussex,
Brighton BN1 9QH, UK
e-mail: `johnca@cogs.susx.ac.uk`

ANN COPESTAKE
Center for the Study of Language and Information, Stanford University,
Stanford, CA 94305, USA
e-mail: `aac@csli.stanford.edu`

(*Received 3 November 1999; revised 27 January 2000*)

Abstract

One major obstacle to efficient processing of large wide coverage grammars in unification-based grammatical frameworks such as HPSG is the time and space cost of the unification operation itself. In a grammar development system it is not appropriate to address this problem with techniques which involve lengthy compilation, since this slows down the edit-test-debug cycle. Nor is it possible to radically restructure the grammar.

In this paper we describe novel extensions to an existing efficient unification algorithm which improve its space and time behaviour (without affecting its correctness) by substantially increasing the amount of structure sharing that takes place. We also describe a fast and automatically tunable pre-unification filter (the 'quick check') which in practice detects a large proportion of unifications that if performed would fail. Finally, we present an efficient algorithm for checking for subsumption relationships between two feature structures; a special case of this gives a fast equality test. The subsumption check is used in a parser (described elsewhere in this volume) which 'packs' local ambiguities to avoid performing redundant sub-computations.

1 Introduction

Unification-based grammatical frameworks such as Lexical-Functional Grammar (Bresnan & Kaplan, 1982) or Head-Driven Phrase Structure Grammar (Pollard & Sag, 1994) have emerged as the dominant linguistic theories for computational applications. Unfortunately, the complexity of these linguistically sophisticated approaches to grammar has made them difficult to use in large-scale applications.

A major obstacle to efficient processing of unification-based grammatical frameworks is the unification operation itself. Roughly 90 per cent of the CPU time expended in parsing a sentence using a large-scale unification based grammar goes into feature structure unification (or copying necessitated by the use of destructive unification, see Section 2). Therefore, any improvements in the efficiency of unification have direct consequences for the overall performance of the system.

There are a variety of approaches to improving efficiency via grammar compilation, as discussed in other papers in this volume for instance. These techniques hold great promise for improving the throughput of a production system, but seem less appropriate for use in a grammar development environment. For development, grammar loading time is as important as sentence processing time, since the most important factor is the time required for the edit-test-debug cycle, and in this context the compilation process itself may be too expensive. Also, debugging will only be possible if compilation is (efficiently) reversible, so that the grammar writer is able to view familiar structures.

Thus, for a development system, what is needed is a way to reduce the cost of unification without lengthy compilation and while still preserving feature structure representations that fit in with a computational linguist's intuitions. In this paper, we describe a number of optimisations that, when taken together, lead to a significant improvement in the performance of the LKB system, a unification-based grammar development, parsing and generation platform originally designed as part of the ACQUILEX project and currently being developed at CSLI (Copestake, 1992, 1999). The version of the typed feature structure formalism assumed in the LKB system is essentially a superset of that given in the appendix to this volume. The main extension to the formalism is that the LKB allows the use of default unification as defined by Lascarides and Copestake (1999). However, in the current paper we will be concerned with purely monotonic processing which is essentially unaffected by the availability of defaults. A somewhat more relevant point is that the LKB system enforces an acyclicity condition on feature structures, though the unification algorithm described in the next two sections can also be used for cyclic feature structures.

2 Graph unification

Unification can be defined in terms of the subsumption ordering of feature structures, as in the appendix to this volume, but computationally, it is easier to think of feature structure unification as *graph unification* of the corresponding DAGs (see Figure 1).

Unification of two DAGs can be carried out in a simple and efficient manner using a variation of the UNION-FIND algorithm (Aho, Hopcroft, & Ullman, 1974; Huet, 1976; Sikkel, 1997). Unification proceeds as in Figure 2. First, the root nodes of the two feature structure graphs are joined into a single *equivalence class*, and one member of the class is selected to be the *representative* of the class (in this case, node 1 is selected to be the representative of the class $\{1, 6\}$). Next, the outgoing edges from all members of the class need to be transferred to the representative. If a

$$\begin{bmatrix} \mathrm{F} & \boxed{1} \\ \mathrm{G} & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} \mathrm{F} & \begin{bmatrix} \mathrm{H} & a \end{bmatrix} \\ \mathrm{G} & \begin{bmatrix} \mathrm{J} & b \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathrm{F} & \boxed{1}\begin{bmatrix} \mathrm{H} & a \\ \mathrm{J} & b \end{bmatrix} \\ \mathrm{G} & \boxed{1} \end{bmatrix}$$

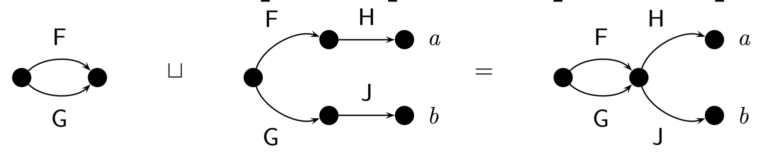

Fig. 1. Feature structure unification and graph unification

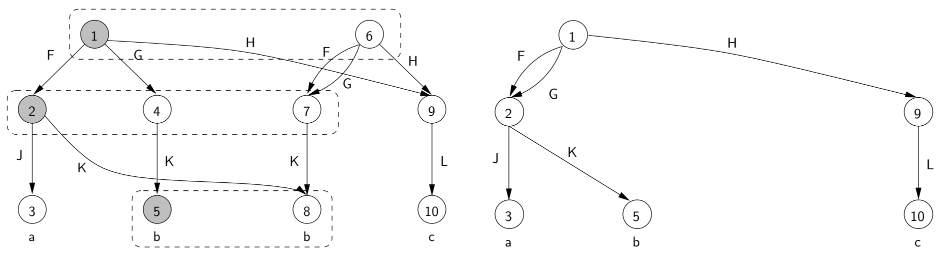

Fig. 2. Destructive graph unification

particular label appears on only one edge, then this edge can simply be copied to the representative. This is the case for the edge labelled H from node 6 to node 9. If the same label occurs more than once as the label of an outgoing edge from a node in a single equivalence class, then the algorithm proceeds recursively, joining the target nodes into a single equivalence class. In the example, the label F appears more than once, so the target classes $\{2\}$ and $\{7\}$ are joined into a class $\{2, 7\}$ (with 2 as the representative). The label G also appears more than once, so the target classes $\{2, 7\}$ and $\{4\}$ are also joined into the class $\{2, 4, 7\}$ (with 2 as the representative). Unification proceeds in this manner until the input graphs have been fully traversed or an incompatibility is found. If unification has completed successfully, the result can be read off by finding the representative of each equivalence class constructed by the UNION-FIND algorithm.

Graph unification is quite simple to implement and is also relatively efficient—it requires very little space beyond what is necessary to store the input feature structure graphs, and its time complexity is almost linear in the size of the input feature structures. Unfortunately, though, this kind of graph unification is *destructive*. At least one of the input feature structure graphs is consumed in the course of the unification, making it unsuitable for use with chart-based parsing and generation schemes as used by the LKB.

As the parser or generator constructs the chart, each edge must be built without modifying the edges that contributed to it. One option to avoid modifying daughter edges during parsing is to copy the input feature structures graphs before performing a destructive unification operation on them. Unification will change the copy of the input structures, leaving the original edge unchanged. However, this will result in many structures being built which never become part of the chart. While

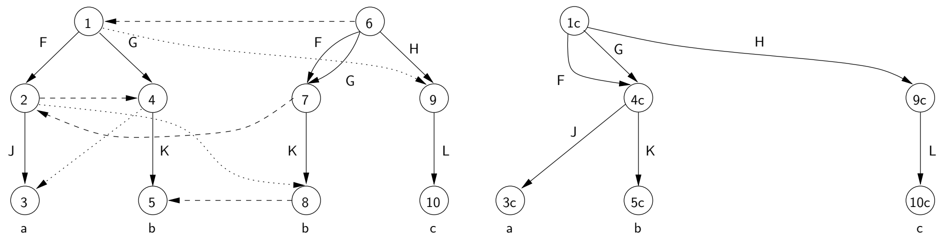

Fig. 3. Quasi-destructive graph unification

parsing with the LinGO English Resource Grammar (described in the introduction to this volume), more than 90 per cent of the unifications that are attempted ultimately fail. If one were to copy the input structures before each unification, then that means that more than 90 per cent of the structures which were built would be discarded without being used.

A more conservative option, taken by Wroblewski (1987), is to apply a non-destructive unification algorithm which constructs a new node to serve as the representative of each equivalence class, rather than using a node from one of the input structures. This has the effect of producing a copy of the result up to the point at which a failure occurs. While this is an improvement over copying the entire structure, either approach will result in some structures being built in the course of an unsuccessful unification, wasting space and reducing the overall throughput of the system.

To avoid these problems, Tomabechi (1991) proposed a *quasi-destructive* variant of the unification algorithm sketched above. Tomabechi's algorithm is based on two observations: copying structures is expensive, and unification usually fails during parsing. Therefore, an algorithm which only copies structures when unification succeeds will be more efficient than one which always copies.

The challenge, of course, is to know when unification will succeed. To avoid copying unnecessarily, Tomabechi's algorithm traverses the input structures to verify that they are compatible. Only when it is guaranteed that unification will be successful is the output structure produced.

Quasi-destructive unification proceeds as in Figure 3. As in Figure 2, nodes 1 and 6 are joined into a single equivalence class. However, this is done non-destructively by setting a temporary forward pointer (indicated by a dashed line) from 6 to 1, the representative of the class. And, again as in Figure 2, the edge corresponding to the feature H is copied from node 6 to node 1. In this case, however, only a temporary edge (indicated by a dotted line) is constructed. Unification proceeds recursively, as before, until the input structures are traversed or a conflict is found. If conflicting values cause unification to fail, the input structures can be returned to their original state simply by invalidating all the temporary forward pointers and edges.[1] Since no new nodes have been built, a failed unification incurs no copying costs. While

[1] This can be done in constant time by marking each temporary pointer with a *generation counter* (Wroblewski, 1987). Pointers are only considered valid if their generation number is the same as the current global generation. When unification fails, the global

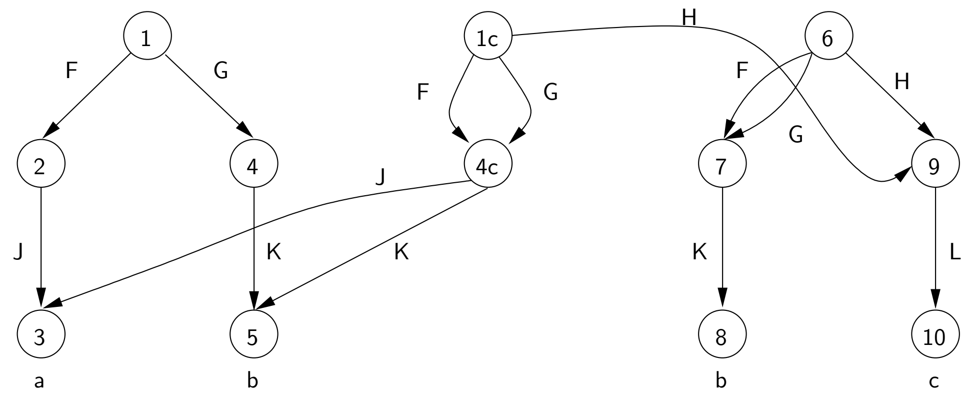

Fig. 4. Subgraph sharing

this method requires constructing temporary pointers whether unification succeeds or not, with careful memory management the space costs of these pointers can be minimised, as Callmeier (this volume) shows. If unification succeeds, on the other hand, then a new output structure needs to be built. This can be done simply by constructing a new node to represent each equivalence class built in the first stage.

3 Subgraph sharing

Tomabechi's unification algorithm is an improvement over simple non-destructive unification when unification fails (which, as we have seen, is most of the time). However, when unification succeeds it builds a complete copy of the result even when parts of the result are identical to parts of the input structure. Compare the output in Figure 3: the nodes 3c, 5c, and 9c are identical to their counterparts in the input. The result could be equivalently constructed as in Figure 4. In this case, the only new nodes that need to be constructed are 1c and 4c. Input nodes 3, 5, and 9 are reused in the output structure, leading to a reduction in the cost of a successful unification compared with Tomabechi's algorithm.

When applied to chart-based parsing or generation, subgraph sharing can yield significant improvements in efficiency. In the construction of the chart, much of the structure associated with an edge is contributed unchanged from the daughter edges. Subgraph sharing means that only a small part of the feature structure needs to be constructed from scratch for each new edge.

In particular, a node in a feature structure graph only needs to be copied if it meets one of four criteria:

1. has any temporary feature pointers
2. is atomic and its value has changed
3. has a descendant which needs to be copied

generation counter is incremented, invalidating all pointers from the earlier computation in a single operation.

4. is part of the grammar

The first three cases are fairly obvious: a new output node has to be constructed when it is not identical to one of the input nodes.

The fourth case however may not be so clear. Feature structures in a parse chart should not share substructures with feature structures from grammar rules or lexical entries. Reusing parts of grammar rules or lexical entries during parsing can lead to spurious cyclic structures and structure sharing. For example, with a grammar which represents coindexing via graph reentrancies, subgraph sharing may lead to multiple occurrences of a pronoun in a sentence like *Chris introduced him to him* being incorrectly coindexed. Even more subtle problems can arise when a single rule is used more than once in the derivation of a single sentence, or when the parser modifies a node through a mechanism other than unification (e.g. by restriction). An inelegant way to avoid this is to instantiate the chart with fresh copies of all rules and lexical entries, and to copy nodes before applying a restriction operator. This extra copying however defeats the purpose of quasi-destructive unification. Instead, we can mark each node in a feature structure graph with a SAFE flag.[2] Nodes built while constructing a parse chart are guaranteed to be unique, so they are SAFE to reuse in subgraph sharing. Nodes built while constructing the grammar, on the other hand, may be introduced more than once during parsing, and so are un-SAFE. Only SAFE nodes are eligible for subgraph sharing.

The algorithm described here implements subgraph sharing in much the same way as the variant of quasi-destructive unification described by Tomabechi (1992). However, there is a crucial difference. Since Tomabechi does not distinguish between SAFE and un-SAFE nodes, his version of structure sharing requires that the parser often make a complete copy of a node before using it. This greatly limits the improvement in efficiency that subgraph sharing can offer. Using the algorithm described here, the parser is only required to make a copy of a feature structure before using it when the same lexical entry is introduced more than once as a daughter of a single chart edge. Fortunately, this situation is easy to detect and almost never occurs with realistic grammars.

On the other hand, the present algorithm is quite different from other approaches to structure sharing found in the literature. Unlike, for example, Pereira's (1985) or Emele's (1991) technique, this implementation of structure sharing adds no overhead to the cost of dereferencing a feature structure. This difference is especially important for a chart-based parser, in which a large number of alternate derivations needs to be explored at once, but in which it is never necessary to undo the effect of more than one unification.[3]

The effect of typed feature structure unification with subgraph sharing on the

[2] Note that the addition of this flag need not increase the size of the data structures or the cost of copying. For example, the SAFE flag may be efficiently represented by a single bit 'packed' into the node data structure, or even, since a node is either SAFE or un-SAFE for its entire lifetime, by using the type system of the underlying programming language rather than part of the data structure itself.

[3] For a more detailed comparison of quasi-destructive unification with other approaches, see Tomabechi (1991, 1992).

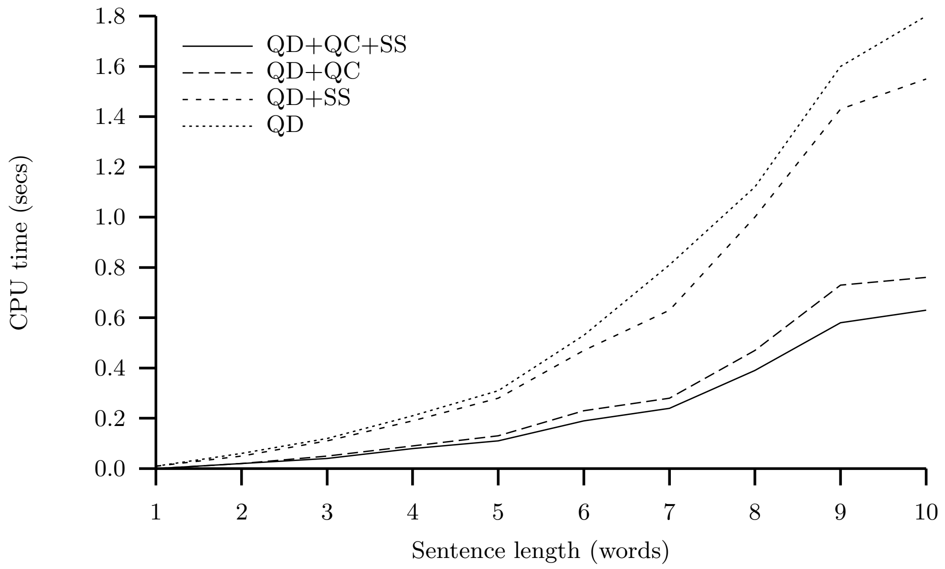

Fig. 5. Mean parse time for items of one to ten words in length from the '*blend*' test suite (see the introduction to this volume), using Tomabechi's quasi-destructive (QD) unification, QD unification with subgraph sharing (SS), QD unification with quick check (QC) filtering using 30 paths, and QD unification with both optimisations. Parse times do not include the cost of garbage collection.

Table 1. *Mean figures for all sentences of one to ten words from the 'blend' test suite.*

	Total time	CPU time (secs)	GC time	Space (MB)
quasi-destructive unification	1.99	0.62	1.37	4.76
QD unification with quick check	1.30	0.27	1.03	3.53
QD with structure sharing	0.91	0.56	0.35	1.44
QD, structure sharing, quick check	0.51	0.23	0.28	0.79

cost of unification when compared to a typed version of Tomabechi's (1991) quasi-destructive unification is shown in Figures 5 and 6 and in Table 1. The figures we give here, and in the rest of this article, are with the LinGO grammar and the LKB system, running in Franz Allegro Common Lisp 5.0 (Linux) on a 450MHz Pentium PC with 512 megabytes of memory. The figures were collected using [incr tsdb()] (Oepen & Flickinger, 1998). Subgraph sharing alone brings a modest reduction in CPU times but a more dramatic reduction in the amount of space required. We have omitted garbage collection (GC) times in Figure 5 because global garbage collections cannot be accurately allocated to sentences of a given length, but we include them in Table 1. As this shows, the reduction in space used brings an

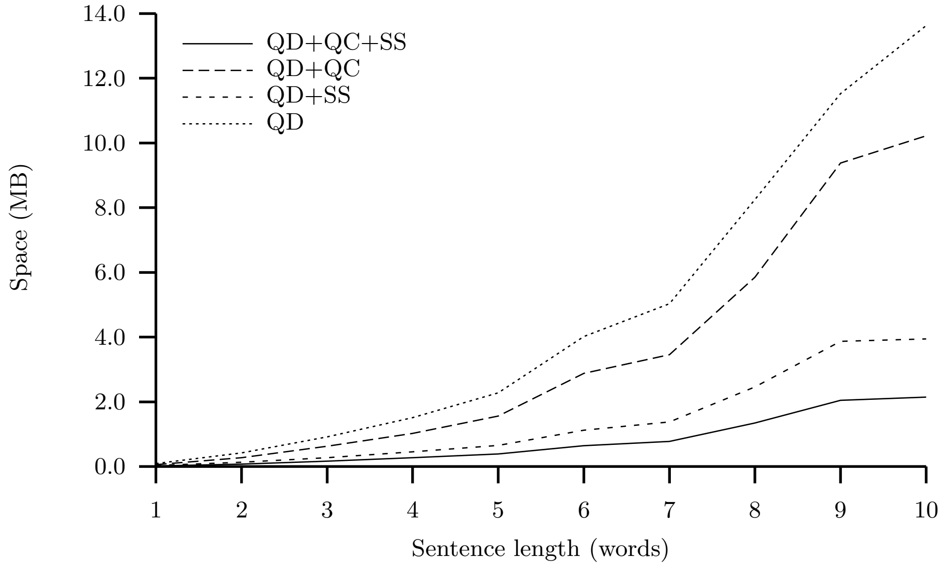

Fig. 6. Mean space used parsing items of one to ten words in length from the 'blend' test suite, using Tomabechi's quasi-destructive (QD) unification, QD unification with subgraph sharing (SS), QD unification with quick check (QC) filtering using 30 paths, and QD unification with both optimisations.

even greater improvement in parser throughput than Figure 5 would suggest. And, when combined with the 'quick check' filter described in Section 5, subgraph sharing brings an even more significant improvement in both space and time efficiency.

4 Type operations

We have described the unification algorithms as though they were operating on untyped feature structures, but the LKB system and the LinGO grammar both assume that structures are typed. This means that when combining nodes their types have to be compared, with the type on the resulting node being the greatest lower bound of the types on the input nodes. Although this operation is often implemented by table lookup, this technique is problematic for grammars that contain a large number of types, such as the LinGO grammar, both because of the size of the table and the time taken to compute it. Our solution rests on the observation that the vast majority of type pairs are never compared, which means that during processing we can use a relatively expensive operation on the uncompiled type hierarchy combined with memoisation to cache the results as they occur. This is a technique that has been known for some time,[4] but we mention it here to emphasise there is no appreciable runtime performance cost compared with full table lookup, at least with

[4] We are grateful to an anonymous reviewer for informing us that it is discussed in the context of the CUF system by Dorna (1992).

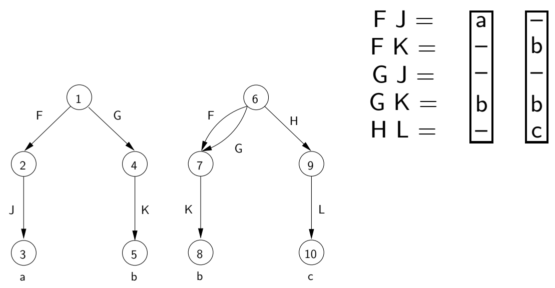

Fig. 7. Deriving quick check vectors.

the LinGO grammar. For instance, around 98.5 per cent of type pairs compared are found in the cache when parsing a five word sentence even when starting with an empty cache. Starting with a perfect cache as opposed to an empty cache caused no measurable difference in processing time. After parsing the first three sentences from a randomised version of the 'blend' test suite, the hit rate was 99.6 per cent — it remained at roughly this level in subsequent sentences (cache misses mostly being caused by semantic types introduced in newly encountered lexical entries).

A second issue in typed feature structure unification is maintenance of well-formedness with respect to a type system (as defined in the appendix to this volume). To maintain well-formedness, it is necessary to consider the constraint on a type just in case the constraint is not equal to the unification of the constraints on the types being compared. This situation requires an extra unification step to maintain well-formedness, but this occurs in such a small percentage of unifications in the LinGO grammar that it does not seriously impact performance.

5 Pre-unification filtering: the 'quick check'

Subgraph sharing minimises the amount of structure that needs to be built when unification succeeds, and quasi-destructive unification avoids unnecessary structure being built when unification fails. However, determining whether unification will succeed or fail still potentially requires complete traversal of the input feature structures. What is missing is a way to quickly identify the majority of cases where unification will fail.

Recall that a feature structure can be represented either as a graph or as a set of path value and path equivalence constraints. In order for two feature structures to be unifiable, all of their path value constraints must be compatible. This observation provides us with a simple mechanism for filtering out unifications that cannot possibly succeed. While processing a sentence, in addition to representing each feature structure used as a graph, we also derive from and associate with the feature structure a vector of path values (which we call the 'quick check' vector), as in Figure 7. The vector can be derived efficiently, and is computed once only for any given feature structure. Before attempting to unify two feature structures, we verify that corresponding values in the respective quick check vectors are compatible. If any pair is incompatible we do not carry out the unification since it is guaranteed to fail.

```
FUNCTION collect-paths-using-counting (failure-paths,n);
  sorted ← sort failure-paths on |failures(failure-path)| in descending order;
  return(first n elements of sorted);
END;

FUNCTION collect-paths-using-discounting (failure-paths,n);
  paths ← empty;
  UNTIL (failure-paths is empty OR n = 0) DO
    best ← element of failure-paths maximising |failures(failure-path)|;
    IF (|failures(best)| = 0) THEN                          {no more unification failures?}
      return(paths);
    failure-paths ← failure-paths ⊖ best;                  {remove best from failure-paths}
    paths ← paths ⊕ best;                                   {add best to end of paths list}
    n ← n − 1;
    FOR EACH path IN failure-paths DO                 {remove best's unification failures
      failures(path) ← failures(path) ⊖ failures(best);       from all other failure-paths}
  return(paths);
END;
```

Fig. 8. Two quick check path collection algorithms.

If all paths were equally likely to cause unification failure then this pre-unification check would not yield much of an improvement. Fortunately, in practice it turns out that the vast majority of unification failures are caused by a relatively small number of paths. For example, features encoding syntactic information such as part of speech are very frequent points of failure, whereas unification never fails on semantic features which are used merely to accumulate portions of the logical form. Since all substructures are typed, unification failure is manifested by a failure to find a greatest lower bound when attempting a type comparison. So, in the quick check vector we store the (type) values at the end of each of a pre-determined small, fixed set of paths that cause unification to fail most often. The paths are ordered from most to least frequently failing. This pre-unification check allows us to filter out a large percentage of failed unifications in constant time.

To find a set of suitable quick check paths we perform an offline computation, processing a set of sentences using a version of the normal unification engine modified so that it does not return immediately after the first type comparison failure, but instead records in a global data structure all paths at which failure occurred, each with an identifier that represents the unification attempt itself.[5] Figure 8 shows two alternative algorithms for computing from the set of paths that failed (*failure-paths*) a set of n quick check paths. The first, more straightforward, algorithm returns the n paths with the highest total counts for failed unifications (calling *failures(path)* to obtain the set of identifiers for unification attempts that failed for a given path). However, this approach is in general not optimal. We want the minimal set of paths that are responsible for the maximum number of unification failures, but this algorithm might return more than one path accounting for the exact same failure.

[5] We implement this as a counter which is incremented before every unification attempt.

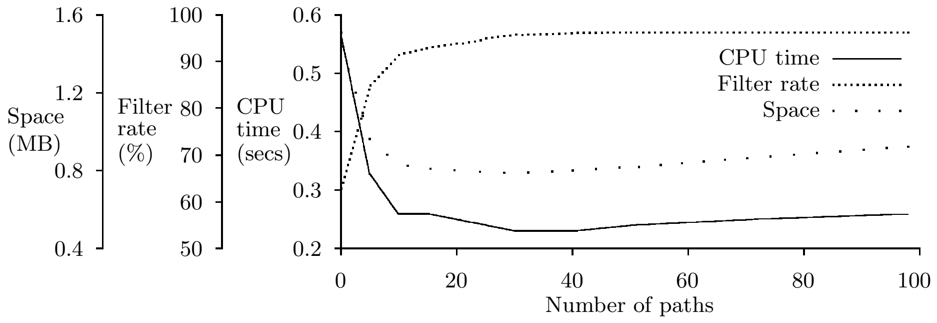

Fig. 9. Mean parse time, filter rate, and space used parsing items of one to ten words in length from the '*blend*' test suite, with the discounting quick check path collection algorithm and varying the numbers of quick check paths collected. The paths were computed from a 300-item random sample of the '*csli*' test suite.

The problem is fixed in the second algorithm, which collects the paths with the highest total counts one by one, for each one *discounting* in all remaining paths the unification failures it covers.

Clearly, when considering the number of paths n required there is a tradeoff between the time savings from filtered unifications and the effort required to create the vectors and compare them. The main factors involved are the relative speeds of type comparison and feature structure unification[6] and the percentage of unification attempts filtered out with a given set of paths. The optimum number of paths therefore cannot be determined analytically. With the LKB and the LinGO grammar, and using the discounting algorithm, we obtain the best parsing performance with around thirty paths (Figure 9). The best performance with the counting algorithm is also at around thirty paths, but resulting parse times are around eight per cent slower, and space usage is three per cent higher. The paths derived are somewhat surprising (Table 2), and in many cases do not fit in with grammar writer intuitions. In particular, some of the paths are very long. An optimal set of paths for a grammar of this complexity could not be produced manually. Although path collection is more expensive than a normal parsing or generation run[7] it only needs to be re-done when the grammar has changed significantly.

The quick check technique will only be of benefit if type comparison is computationally cheap — as indeed it is in our implementation (Section 4) — and if the filter rate is high (otherwise the extra work performed essentially just duplicates work carried out later in unification). We also apply a simple statically-computed

[6] To further reduce the time spent in quick check vector compatibility tests, type compatability checking is replaced by a single logical *and* bit-vector operation in the case of paths whose maximal types have only a small number of descendents.

[7] We have found that the sets of paths collected differs little between our parsers (Oepen & Carroll, this volume) and (chart-based) generator (Carroll, Copestake, Flickinger, & Poznanski, 1999).

Table 2. *First few quick check paths and associated frequencies.*

Path	Frequency (discounted)
SYNSEM LOCAL CAT HEAD	247123
INFLECTED	125914
SYNSEM LOCAL CAT VAL COMPS FIRST OPT	65262
SYNSEM LOCAL CAT VAL COMPS	52893
SYNSEM LOCAL CAT HEAD VFORM	36799
SYNSEM LOCAL CAT HEAD MOD	36608
SYNSEM LOCAL KEYS KEY	33576
SYNSEM LOCAL CAT MC	21762
SYNSEM NON-LOCAL QUE	21113
SYNSEM LOCAL CAT HEAD MOD FIRST LOCAL CAT HEAD	17137
SYNSEM LOCAL CONT MESSAGE	16553
SYNSEM NON-LOCAL SLASH LAST	11683
SYNSEM	9383
SYNSEM LOCAL CAT HEAD MOD FIRST LOCAL	8353
SYNSEM LOCAL CAT VAL SPR FIRST	7938

filter on rule application (described by Kiefer, Krieger, Carroll, & Malouf, 1999) in conjunction with the quick check. Although the quick check is the more powerful filter of the two because it functions *dynamically*, taking into account feature instantiations that occur during the parsing process, the rule filter is still valuable when executed first since it consists of a single, very fast table lookup. With the LinGO grammar the overall filter rate is around 96 per cent. Thus almost all failing unifications are identified quickly without incurring the cost of full unification.

Kogure (1990) outlines a related idea to the quick check, designed to detect failing unifications quickly (but unfortunately he does not quote any performance figures). In this approach information is acquired in a training phase about which features introduced by which types most frequently lead to failed unifications. This is done by randomising the order in which the features occur in feature structures and recording the failure point; then arcs are sorted dynamically during the course of each unification to arrange that these features are considered first. (A similar technique was developed, independently, by Backofen and Krieger (1993)). However, the quick check technique has a number of advantages over this approach, since it guarantees to collect *all* failures in the training phase (so more reliable information is collected from less training data), complete global paths are recorded rather than individual feature-type pairs (resulting in finer tuning to the actual points of failure and avoiding even entering the unification procedure for unifications which are bound to fail), and a time-consuming sort operation at each feature structure node is not required during unification.[8]

There is also an interesting relationship between the set of paths computed by the quick check and the construction of context-free (CF) backbones from unification

[8] A number of typed-unification systems have provided facilities for manually specifying a set of paths to be checked before unification is attempted; the quick check extends, optimises, and automates the technique.

grammars (see Torisawa, Nishida, Miyao, and Tsujii, this volume). However, while approaches to CF backbone computation perform a static analysis of the grammar, quick check path computation is dynamic since it is based on data collected during the processing of a training corpus. It would be interesting to to see if the dynamic approach could be used in CF backbone construction. We also note that applying the quick check in a system that also used CF pre-filtering is unlikely to be as effective as in our experiments since the quick check paths and the CF backbone encode similar kinds of information.

6 Efficient subsumption and equality checking

The most important operation when processing with unification based grammatical frameworks such as HPSG is the unification operation itself. However, in parsers that perform *local ambiguity packing* (see Oepen & Carroll, this volume), a second, central operation is subsumption checking (defined in the appendix to this volume).

The feature structure subsumption algorithm we describe here[9] employs similar machinery to the quasi-destructive unification algorithm. In particular, it uses temporary pointers to keep track of intermediate results in processing, in conjunction with a generation counter which is incremented at the end to invalidate all pointers in a single operation. But whereas the unification algorithm makes two passes—an initial traversal of both feature structures, followed by a second traversal to make a copy if the unification is successful—the subsumption algorithm makes only one pass, checking reentrancies and type-supertype relationships at the same time. The algorithm also simultaneously checks if either feature structure subsumes the other, or if there is no subsumption relation between them in either direction.

Figure 10 shows the algorithm. The top-level entry point *dag-subsumes-p* and subsidiary function *dag-subsumes-p0* each return two values, held in variables *forwardp* and *backwardp,* both initially true, recording whether it is possible that the first dag subsumes the second and/or vice-versa, respectively. When one of these possibilities has been ruled out the appropriate variable is set to false. (In the statement of the algorithm the two returned values are notated as a pair, i.e. *(forwardp,backwardp)*). If at any stage both variables have become set to false the possibility of subsumption in both directions has been ruled out so the algorithm exits.

The (recursive) subsidiary function *dag-subsumes-p0* does most of the work, traversing the two input dags in step. First, it checks whether the current node in either dag is involved in a reentrancy that is not present in the other: for each node visited in one dag it adds a temporary pointer (held in the ‘copy’ slot) to the corresponding node in the other dag. If a node is reached that already has a pointer then this is a point of reentrancy in the dag, and if the pointer is not identical to the other dag node then this reentrancy is not present in the other dag. In this

[9] Although independently-developed implementations of essentially the same algorithm can be found in the source code of The Attribute Logic Engine (ALE) version 3.2 (Carpenter & Penn, 1999) and the SICStus Prolog (SICS, 1999) term utilities library (Penn, personal communication), we believe that there is no previous published description of the algorithm.

```
FUNCTION dag-subsumes-p (dag1,dag2);
   (forwardp,backwardp) ← catch with tag subsume-fail
                              calling dag-subsumes-p0(dag1,dag2,true,true);
   increment *unify-global-counter*;                   {invalidate temporary pointers}
   return(forwardp,backwardp);
END;

FUNCTION dag-subsumes-p0 (dag1,dag2,forwardp,backwardp);
   IF (dag1.copy is empty) THEN                                    {check reentrancies}
      dag1.copy ← dag2;
   ELSE IF (dag1.copy ≠ dag2) THEN
      forwardp ← false;
   IF (dag2.copy is empty) THEN
      dag2.copy ← dag1;
   ELSE IF (dag2.copy ≠ dag1) THEN
      backwardp ← false;
   IF (forwardp = false AND backwardp = false) THEN         {reentrancy check failed?}
      throw(false,false) with tag subsume-fail;
   UNLESS supertype-or-equal-p(dag1.type,dag2.type) THEN                {check types}
      forwardp ← false;
   UNLESS supertype-or-equal-p(dag2.type,dag1.type) THEN
      backwardp ← false;
   IF (forwardp = false AND backwardp = false) THEN               {type check failed?}
      throw(false,false) with tag subsume-fail;
   IF (has-arcs-p(dag1) AND has-arcs-p(dag2)) THEN                        {check arcs}
      shared ← intersectarcs(dag1,dag2);
      FOR EACH arc IN shared DO
         (forwardp,backwardp) ← dag-subsumes-p0(destination of shared arc for dag1,
                                                destination of shared arc for dag2,
                                                forwardp,backwardp);
   return(forwardp,backwardp);
END;
```

Fig. 10. The feature structure subsumption algorithm.

case the possibility that the former dag subsumes the latter is ruled out. After the reentrancy check the type-supertype relationship between the types at the current nodes in the two dags is determined, and if one type is not equal to or a supertype of the other then subsumption cannot hold in that direction. Finally, after successfully checking the type-supertype relationships, the function recurses into the arcs outgoing from each node that have the same label. Since we are assuming totally well-typed feature structures (Carpenter, 1992), it must be the case that either the sets of arc labels in the two dags are the same, or one is a strict superset of the other. Only arcs with the same labels need be processed; extra arcs need not since the type-supertype check at the two nodes will already have determined that the feature structure containing the extra arcs must be subsumed by the other, and they merely serve to further specify it and cannot affect the final result.

The function *intersectarcs(dag1,dag2)* is the same as that of Tomabechi and Wroblewski (1987), returning the arcs with labels that exist in both *dag1* and *dag2*;

supertype-or-equal-p(type1,type2) succeeds if *type1* is either equal to or a supertype of *type2*; and *has-arcs-p(dag)* succeeds if the dag has any arcs. Our implementation of the algorithm contains extra redundant but cheap tests that give a slight speed improvement but which for reasons of clarity are not shown in Figure 10: a test for equality of the two node types which if successful circumvents the supertype checking portion of the code, and tests that *forwardp* is true immediately before the first *supertype-or-equal-p* check and that *backwardp* is true before the second[10].

The use of temporary pointers means that the space complexity of the algorithm is linear in the sum of the sizes of the feature structures. However, in our implementation the 'copy' slot that the pointers occupy is already present in each dag node (it is required for the final phase of unification to store new nodes representing equivalence classes), so in practice the subsumption test does not allocate any new storage. Assuming the supertype tests can be carried out in constant time (e.g. by table lookup), and that the grammar allows us to put a small constant upper bound on the intersection of outgoing arcs from each node, the processing in the body of *dag-subsumes-p0* takes unit time. The body may be executed up to N times where N is the number of nodes in the smaller of the two feature structures. So overall the algorithm has linear time complexity. Running on the processing platform specified in Section 3, our implementation performs of the order of 34,000 feature structure subsumption tests per second, when applied to all pairs of constituents covering the same words in parses of all items of one to ten words in length from the '*blend*' test suite (without any local ambiguity packing). Approximately 18 per cent of constituents were found to be related by subsumption.

The fact that the subsumption algorithm simultaneously and efficiently checks whether either input feature structure subsumes or is subsumed by the other gives us an easy way to construct an efficient algorithm for checking if two feature structures are *equal*. The algorithm invokes *dag-subsumes-p* on the two input feature structures, and if the two results *forwardp* and *backwardp* are both true (i.e. mutual subsumption holds) then the feature structures are equal. Uses for the equality test might include testing for duplication of analyses produced by a parser, perhaps as part of a debugging tool in a grammar development environment; or as a weaker alternative to the subsumption test, as used in some approaches to local ambiguity packing (Miyao, Makino, Torisawa, and Tsujii, this volume).

[10] There is scope for further optimisation of the algorithm in the case where *dag1* and *dag2* are identical: full processing inside the structure is not required (since all nodes inside it will be identical between the two dags and any strictly internal reentrancies will necessarily be the same), but we would still need to assign temporary pointers inside it so that any external reentrancies into the structure would be treated correctly. In tests with the LinGO grammar we have found that as far as constituents that are candidates for local ambiguity packing are concerned there is in fact little of this type of structure sharing between them, so the special equality processing is not worth the extra complication.

7 Conclusions

Linguistically sophisticated grammatical frameworks rely heavily on feature structure unification. However, without an efficient unification operation such frameworks are impractical for real-world computational applications, or indeed even for development of large-scale grammars.

In this paper we have described a number of optimisations that make unification more efficient. Destructive graph unification can be implemented efficiently, but is not suited to chart-based parsing or generation. Tomabechi's quasi-destructive unification algorithm is well-suited, and avoids unnecessary structure building in the majority of cases when unification fails. We have showed how the algorithm can be made even more efficient by making use of subgraph sharing to minimise the new structure that needs to be built when unification succeeds.

Since unification usually fails, we can further improve performance by making unification fail as early as possible. The 'quick check' mechanism allows us to filter out in constant time a large percentage of unifications that are bound to fail. In effect, this gives us some of the performance benefits of a context-free backbone without changing the way the grammar is represented. The technique has also recently been adapted to give a fast filter on subsumption checking (Kiefer & Krieger, 2000).

We have also devised linear-time algorithms for feature structure subsumption and equality checking. Feature structures that span the same segment of the input string and that are in a subsumption relationship can be dynamically merged into a single entity for further processing. A parser can thus avoid repeatedly performing similar computations on the same portion of the input.

On-line feature structure operations can be made relatively efficient. The techniques described in this paper are sufficient to meet the fairly modest performance demands of a grammar development environment. Coupled with the compilation techniques described elsewhere in this volume for a run-time system and the ever-increasing computational power available to consumers, they may make large-scale linguistically sophisticated grammars efficient enough for real-world applications.

8 Acknowledgements

For useful discussions about the issues addressed here the authors would like to thank three anonymous reviewers, along with Ulrich Callmeier, Dan Flickinger, Bernd Kiefer, and Guido Minnen. We are especially grateful to Stephan Oepen who helped devise the subsumption algorithm, and Gerald Penn for information about other implementations of it. This material is based in part on work supported by the National Science Foundation under grant number IRI-9612682 to CSLI. The first author is supported by a grant from the University of Groningen to the School of Behavioral and Cognitive Neurosciences for the project "Neurobiological control in human performance," and the second author is supported by a UK EPSRC Advanced Fellowship.

References

Aho, A., Hopcroft, J., & Ullman, J. (1974). *The design and analysis of computer algorithms.* Reading, MA: Addison-Wesley.

Backofen, R., & Krieger, H.-U. (1993). The TDL/UDiNe system. In R. Backofen, H.-U. Krieger, S. Spackman, & H. Uszkoreit (Eds.), *Report of the EAGLES workshop on implemented formalisms* (pp. 67–74). DFKI Research Report D-93-27.

Bresnan, J., & Kaplan, R. M. (1982). Lexical-Functional Grammar: a formal system for grammatical representation. In J. Bresnan (Ed.), *The mental representation of grammatical relations.* MIT Press.

Carpenter, B. (1992). *The logic of typed feature structures.* Cambridge University Press.

Carpenter, B., & Penn, G. (1999). *ALE: The Attribute Logic Engine / user's guide version 3.2.* (University of Tübingen, <`http://whorf.sfs.nphil.uni-tuebingen.de/~gpenn/ale/files/guide.tex`>)

Carroll, J., Copestake, A., Flickinger, D., & Poznanski, V. (1999). An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of the 7th European workshop on natural language generation (EWNLG'99)* (pp. 86–95). Toulouse, France.

Copestake, A. (1992). The ACQUILEX LKB: representation issues in semi-automatic acquisition of large lexicons. In *Proceedings of the 3rd conference on applied natural language processing* (pp. 88–96). Trento, Italy.

Copestake, A. (1999). *The (new) LKB system.* (Center for the Study of Language and Information, Stanford University)

Dorna, M. (1992). *Erweiterung der constraint-logiksprache CUF um ein typsystem.* Diplomarbeit no.896, Institut für Informatik, Universität Stuttgart.

Emele, M. C. (1991). Unification with lazy non-redundant copying. In *Proceedings of the 29th annual meeting of the Association for Computational Linguistics* (pp. 323–330). Berkeley, CA.

Huet, G. (1976). *Résolution d'equations dans les langages d'order* $1, 2, \ldots, \omega$. Thèse de Doctorat d'Etat, Université Paris VII.

Kiefer, B., & Krieger, H.-U. (2000). A context-free approximation of Head-driven Phrase Structure Grammar. In *Proceedings of the 6th international workshop on parsing technologies (IWPT 2000)* (pp. 135–146). Trento, Italy.

Kiefer, B., Krieger, H.-U., Carroll, J., & Malouf, R. (1999). A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics* (pp. 473–480). College Park, Maryland.

Kogure, K. (1990). Strategic lazy incremental copy graph unification. In *Proceedings of the 13th international conference on computational linguistics (COLING-90)* (pp. 223–228). Helsinki, Finland.

Lascarides, A., & Copestake, A. (1999). Default representation in constraint-based frameworks. *Computational Linguistics*, *25*, 55–106.

Oepen, S., & Flickinger, D. (1998). Towards systematic grammar profiling. Test suite technology ten years after. *Computer Speech and Language*, *12(4)*, 411–436.

Pereira, F. (1985). A structure sharing representation for unification-based grammar formalisms. In *Proceedings of the 23th annual meeting of the Association for Computational Linguistics* (pp. 137–144). Chicago, IL.

Pollard, C., & Sag, I. A. (1994). *Head-driven Phrase Structure Grammar.* Chicago and Stanford: University of Chicago Press and CSLI Publications.

SICS. (1999). *SICStus Prolog user's manual release 3.8.* (Intelligent Systems Laboratory, Swedish Institute of Computer Science, Sweden, <`http://www.sics.se/sicstus/`>)

Sikkel, K. (1997). *Parsing schemata.* Berlin, Germany: Springer-Verlag.

Tomabechi, H. (1991). Quasi-destructive graph unification. In *Proceedings of the 29th annual meeting of the Association for Computational Linguistics* (pp. 315–322). Berkeley, CA.

Tomabechi, H. (1992). Quasi-destructive graph unification with structure-sharing. In *Proceedings of the 14th international conference on computational linguistics (COLING-92)* (pp. 440–446). Nantes, France.

Wroblewski, D. A. (1987). Non-destructive graph unification. In *Proceedings of the 6th national conference on artificial intelligence (AAAI'87)* (pp. 582–587). Seattle, WA.