Technical Report CS-2002-01

A Graph Unification Machine for NL Parsing

Vlado Kešelj and Nick Cercone

Department of Computer Science, University of Waterloo,

Waterloo, ON N2L 3G1, Canada,

`vkeselj` *or* `ncercone` `@cs.uwaterloo.ca`

`http://www.cs.uwaterloo.ca/ ~vkeselj` *or* `~ncercone`

January 23, 2002

Abstract

A simple, novel, and efficient computational model for a graph unification method for NL parsing is presented. We rely the body of existing research on labeled graph unification for natural language parsing. This model offers several advantages including: simplicity, efficiency, and amenability to a low-level, efficient, and straight-forward implementation. A consequence of this is that some earlier considerations with respect to garbage collection and redundant node copying become obsolete. The model uses a novel feature of sub-node structure sharing.

Key words: Parsing, unification.

1 Introduction

The various grammar formalisms for natural languages (NL), such as HPSG (Head-driven Phrase Structure Grammar), LFG (Lexical Functional Grammar), and PATR-II, use labeled graph unification to express grammar constraints and to capture other NL phenomena. We describe in section 2 how graph unification is used to describe NL rules, and how it is implemented at a conceptual level in actual parsers.

A sufficiently efficient algorithm for unifying two graphs is known. However, it is destructive to the argument graphs, which is an undesirable effect in NL parsing. Coping argument graphs is a very expensive solution ([27]), so several algorithms which reduce the total cost of unifications during parsing have been proposed. These algorithms are described in the review of the known techniques and related work in section 3.

The essential ideas of the previous algorithms are sometimes obscured by several unimportant issues, such as considerations with respect to garbage collection and expensive system calls for memory allocation. The algorithms were presented as high-level recursive algorithms, without including all details, such as handling the sets of edges. Memory management is not directly handled, so it can lead to memory fragmentation. Some data fields, such as 'status' and 'mark' fields, are redundant, when the algorithm is re-designed. Motivation for our model with respect to the previous approaches is further described in section 4.

Our model is described in section 5, and in section 6 we give a detailed example. Discussion of some of the remaining issues is given in section 7, and the conclusion is given in section 8. The source code for a C and a Java implementation of the algorithm is given in the appendix, and it is available on the Web.[1].

In this paper, under the term *graph* we assume a labeled directed graph (with labeled nodes and edges). Under the term *rooted graph* we assume a labeled directed graph with a distinguished node, called *root,* such that any other node is reachable from the root node.

2 Graph unification in NL parsing

Let us consider the following sentence:

> The red book is on the table.

If we use a context-free grammar to parse it, then the parse tree shown in figure 1 can be a typical result (without the dotted line). This representation does not capture some syntactic phenomena, such as *agreement.* For example, we cannot say "*The red book are on the table." because the noun phrase and the verb phrase have to agree in *number,* which is singular for 'book'

[1] `http://www.cs.uwaterloo.ca/~vkeselj`

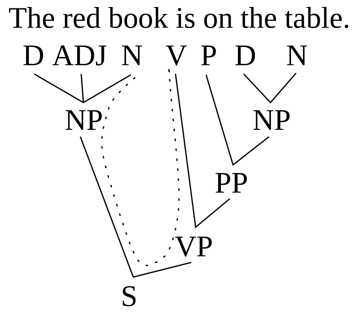

Figure 1: Context-free parse

and plural for 'are'. This can be solved by passing and matching the number information along the path denoted by the dotted line in figure 1.

Issues like these are handled in *unification-based grammars,* like HPSG (Head-driven Phrase Structure Grammar), using *feature-structures* or *attribute-value matrices* (AVM). For the example in figure 1, instead of using lexical rules

$$D \to The, ADJ \to red, N \to book, V \to is, P \to on, D \to the, N \to table,$$

we rewrite words by the following feature structures:

$$\begin{array}{ccccccc} \textit{The} & \textit{red} & \textit{book} & \textit{is} & \textit{on} & \textit{the} & \textit{table.} \\ [\,d\,] & [\,adj\,] & \begin{bmatrix} sm\_object \\ \mathsf{H}\text{: } [\,\mathsf{A}\text{: } [\,\mathsf{N}\text{: } \mathrm{sg}\,]\,] \end{bmatrix} & \begin{bmatrix} be \\ \mathsf{H}\text{: } [\,\mathsf{A}\text{: } [\,\mathsf{N}\text{: } \mathrm{sg}\,]\,] \end{bmatrix} & [\,p\,] & [\,d\,] & [\,n\,] \end{array}$$

The feature structures have *types,* like *d* or *sm_object* above, and they include features, such as H and A above. The features are associated with their values, which can be atoms, such as 'sg', or another feature structures. If a type is not specified, then the *most general type* is assumed.[2]

The types are organized into a type hierarchy. For example, we can specify that *sm_object* is a subtype of *noun*, and *be* is a subtype of *verb*, so that the words 'book' and 'is' above can be used in rules that require types *noun* and *verb*, respectively. The agreement information 'number = singular' is encoded as N:sg, which is part of the agreement (A) structure, which is part of the head (H) information. The head information is passed from the

[2] This description illustrates very briefly the use of feature structures and unification in NL parsing. For a more complete account, see [25, 7].

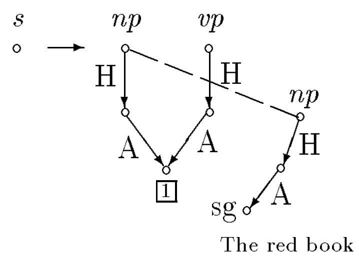

Figure 2: Graph representation: sentence rule and a component

head word to the whole phrase, e.g., by the following two rules:

$$\begin{bmatrix} np \\ \mathsf{H}\text{: } \boxed{1} \end{bmatrix} \rightarrow [d\,][adj\,]\begin{bmatrix} noun \\ \mathsf{H}\text{: } \boxed{1} \end{bmatrix}.$$

$$\begin{bmatrix} vp \\ \mathsf{H}\text{: } \boxed{1} \end{bmatrix} \rightarrow \begin{bmatrix} v \\ \mathsf{H}\text{: } \boxed{1} \end{bmatrix}[pp\,].$$

The agreement between a verb phrase and a noun phrase in a sentence is enforced by the following rule:

$$[s\,] \rightarrow \begin{bmatrix} noun \\ \mathsf{H}\text{: } [\,\mathsf{A}\text{: } \boxed{1}\,] \end{bmatrix}\begin{bmatrix} noun \\ \mathsf{H}\text{: } [\,\mathsf{A}\text{: } \boxed{1}\,] \end{bmatrix}. \tag{1}$$

The indices '$\boxed{1}$' denote the same structures, or *shared structures,* as they are usually called. Shared structures are also called a *reentrancy.*

2.1 Graph representation

A feature structure can be represented as a rooted graph. Hence, a grammar rule can be represented as an array of rooted graphs, which can share structure. For example, the above rule (1) can be represented as graph shown in figure 2. Additionally, the figure illustrates the graph representation of the phrase "The red book," and the dashed line connects two nodes that are unified during parsing. In one of the following steps, a graph representing the phrase "is on the table" is unified with graph rooted at node *vp* and the sentence is parsed. In this example, the parsing result, which we simply call a *parse,* is just the node labeled with *s.* In general a parse is a rooted graph, and it is usually larger than a single node. Some parse information, i.e., some parts of the parse graph, originate from the information encoded in the final rule, and some information is obtained through structure-sharing with the daughter nodes.

2.2 Chart parsing

Chart parsing is a frequently used algorithm for parsing natural languages. A *chart* is a table with entries called *chart edges.* A chart edge covers a continuous sub-string of the sentence, which is specified by its span. The algorithm starts by adding chart edges that cover recognized tokens in the sentence. These edges are *passive edges,* they are part of the *passive chart,* and each of them contains a rooted graph, which is a partial parsing result of the covered sub-string. For each new passive edge, we attempt to unify it with a designated daughter in each rule (e.g., the left-most daughter, or the head daughter). If unification succeeds and the rule has only one daughter, the result is a new passive edge. Alternatively, if unification succeeds and the rule has more daughters, an *active edge* is created, which is added to the *active chart.* Active edges are similar to dotted rules in the Earley's algorithm—some daughters are parsed and they cover a sub-string of the sentence, while one or two daughter nodes surrounding this parsed part are considered for expansion using the passive edges that border the covered span. When an active edge is expanded by unifying one more daughter, the result is either a completed passive edge if all daughters are unified, or a new active edge otherwise.

The process continues until no more edges can be added to the charts. Any passive edge of an appropriate type that covers the whole sentence is a parse. Alternatively, the algorithm can stop earlier as soon as one parse is found.

Unification of the labeled graphs is an operation that uses a significant portion of the running time.

3 Related Work

A useful introduction to unification is the general survey by Knight [14] (1989). It briefly describes some of the work that we review here as well.

Some of the early NL parsers that used graph unification were parsers written for the PATR-II grammar formalism (Shieber [24]). They used an efficient destructive algorithm for labeled graph unification (Pereira [22]). The algorithm is sufficiently efficient with respect to the problem of unifying two graphs; however, it is destructive, i.e., it destroys the original graphs, or at least one of them, whether the unification succeeds or not. In the chart

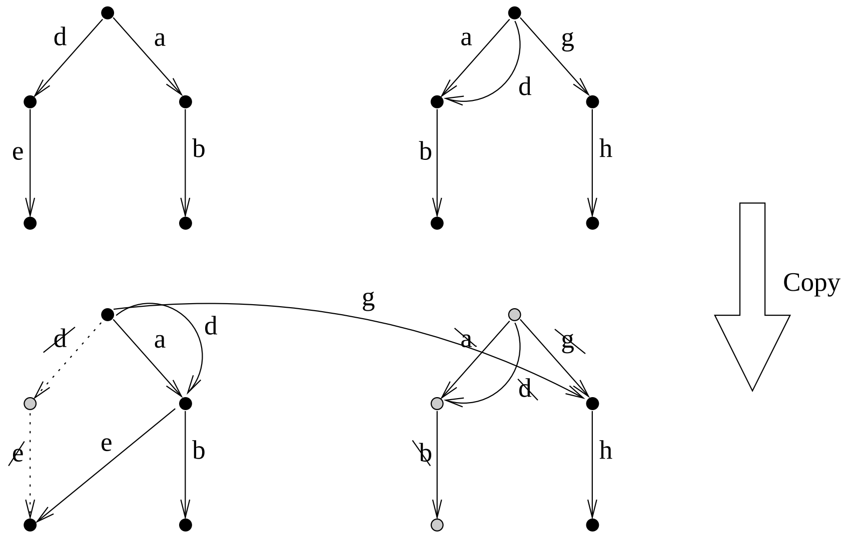

Figure 3: Naïve unification

parsing algorithm, we want to keep the original graphs as well.

In chart parsing, unification is done between a constituent in a rule or in an active edge, and a passive edge. If unification succeeds, then a new edge is created. Whether unification succeeds or not, we want to keep the original rule and the original constituent.

Let us call the original graphs that are unified the *argument graphs,* and the resulting graph if unification succeeds, the *result graph.* A solution is to make copies of the argument graphs before unification, and then to unify the copies. This is a simple and expensive solution, which was used in some early parsers ([24]). Most of the unifications during parsing fails (60% according to Tomabechi [26]), so this excessive copying becomes a "computational sink," as described by Wroblewski [27]. Figure 3 illustrates this simple method with copying, which is called the *naïve method.*[3]

The figure illustrates unification of the following two AVMs represented as graphs:

$$\begin{bmatrix} \mathsf{d}\text{:} \ [\mathsf{e}\text{:} \ [\top]] \\ \mathsf{a}\text{:} \ [\mathsf{b}\text{:} \ [\top]] \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \mathsf{a}\text{:} \ \boxed{1}\,[\mathsf{b}\text{:} \ [\top]] \\ \mathsf{d}\text{:} \ \boxed{1} \\ \mathsf{g}\text{:} \ [\mathsf{h}\text{:} \ [\top]] \end{bmatrix}$$

As we can see from the figure, we first copy all 10 nodes with edges, and then

[3] The example was used in Wroblewski [27].

do destructive unification. There are 6 nodes at the end of unification.

In the process, we did some unnecessary copying. This problem was described by Wroblewski [27], and he differentiates two kinds of unnecessary copying in this approach:

over-copying: When two argument graphs are copied, then too many nodes are copied since the resulting graph has less nodes that the total number of nodes of the two argument graphs. For example, in figure 3 we copied 10 nodes and 9 edges, while the resulting graph used only 6 nodes and 6 edges.

early copying: The problem of early copying is that we make copies in advance, without knowing whether the unification will succeed or not. A better approach would copy as unification proceeds. At a point where unification fails, we stop copying and do not make any extra copying.

Wroblewski [27] describes an algorithm that partially solves this problem: It reduces over-copying, and eliminates early copying. The algorithm does not copy the argument graphs. Instead, it works non-destructively on the original argument graphs. Whenever a change on a node is required, it does not change the original node but makes a copy of it, and changes the copy. Once a copy is created, all additional changes are done on it. A copied object is found by following the *forward* pointer from the original object. There can be a chain of the forward pointers. Whether the unification fails or not, after the procedure the forward pointers from the original nodes to the copied nodes are invalidated using a simple global counter trick, which we will explain later. The effects of the Wroblewski algorithm on the graphs in figure 3 are illustrated in figure 4. The dashed lines depict forward pointers to the created copies. The copies are made as unification progresses, so early copying is eliminated. Over-copying can still occur. For example, during unification, we may unify two nodes A and B, and a copy of the future node A_1 is created. Two other nodes C and D are unified, and a copy C_1 is created for them. However, we may later need to unify A and C. It is done on one of the copies, e.g., A_1, so the copy C_1 was unnecessarily created, and that represents over-copying.

Wroblewski algorithm was a relatively simple solution to the problem, as a modification of the destructive unification algorithm; but it was not the first offered solution. Two solutions were given before by Pereira [22] and Karttunen [13, 12].

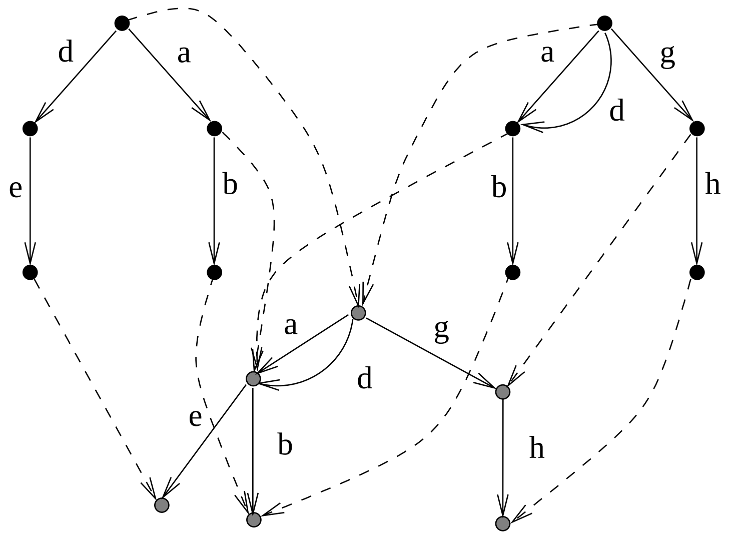

Figure 4: Wroblewski unification

Pereira [22] proposed a structure-sharing method inspired by Boyer and Moore [4] method used in Prolog implementations. During unification, the changes are recorded in an "environment," so the unification is non-destructive. An advantage of this approach is the use of *hidden structure-sharing,* which will be described later in the context of a similar approach by Emele [10]. The method does prevent over-copying and early copying. The disadvantage is that an overhead cost of $O(\log d)$, where d is the number of nodes, is associated with each access to a node. Another disadvantage is that this is a complicated method to implement (Wroblewski [27]).

Karttunen [12] proposed a *reversible unification* method, in which all changes are done on argument graphs, but they are recorded so they can be undone. If unification succeeds, then the result graph is copied and the argument graphs are reversed to their previous state. This method also prevents over-copying and early copying, but the final reversal of all changes and copying in case of a successful unification have significant cost, and these operations are avoided in some other methods.

Global counter trick. Wroblewski [27] used a global counter trick to invalidate node changes on the original nodes after the unification is finished. Suppose that we have a node n, and the field $n.forward$ is used as

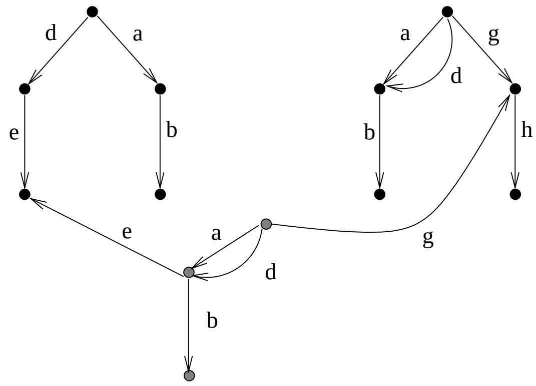

Figure 5: Unification without redundant copying

a pointer to its copy. We also have an integer field *n.generation*, whose value is equal to the current global generation counter *Generation*. Let us assume *Generation* = 1. After finishing unification, whether it succeeds or not, we increase *Generation* to 2, which invalidates all forward pointers. Namely, when the next unification is performed, we will find that $n.generation < Generation$, which means that the forward pointer is not valid and the node n does not have a copy. When a new copy is created, we make the update $n.generation \leftarrow Generation$.

The global counter trick is advanced by Emele [10].

In 1990, Godden [11] describes another technique for lazy unification using *closures,* which are programming language constructs. In the same year, Kogure [15] presents a unification method with lazy incremental copying, and a method that orders the unification of nodes in such way, so that the paths that fail more frequently are unified first.

Redundant copying and hidden structure sharing. Beside over-copying and early copying, Emele [10] defines *redundant copying.* In the previous example in figure 4, we can see that the branch g : h was unnecessarily copied, since it was not changed. This is referred to as the *redundant copying* (Emele [10]). A solution without redundant copying would produce a graph as shown in figure 5. The nodes on the paths g, g:h, and d:e are not changed so we can reuse them in the new graph. The reused structures are shared between different AVMs, so this approach is called *structure sharing.*

We have already introduced the term 'structure sharing' in a different context, as intended sharing of constituents in grammar design (it is also called *reentrancy*). Although both of these senses of 'structure sharing' are very close, and usually implemented in the same way, there is a crucial difference between them. If two paths are structure shared by grammar design, where they are usually part of an graph or a rule, then any change of the sub-graph at the end of one path is reflected in the other path too. However, if two paths are structure-shared in our new sense, i.e., they are shared for the efficiency reasons but they are different at the grammar level, then if the node at one path is changed it has to be separated from the node at the other path before changing it, and they are not 'structure-shared' any more. For this reason, we call this new concept of structure sharing *hidden structure sharing.* There are no established terms for these two kinds of structure sharing. The first kind of structure sharing was called *reentrancy, feature-structure sharing*, and *structure sharing.* The second kind, i.e., *hidden structure sharing,* was called *data-structure sharing*, *subgraph sharing,* and *structure sharing* (Callmeier [6]).

We are not discussing reentrancies here, so under the term *structure sharing* we assume hidden structure sharing.

Hidden structure sharing does not only save time since no redundant copying is done, it also saves memory. These savings can be significant in chart parsing. For example, a daughter constituent can be included in its mother node with no additional copying. In chart parsing, various mother nodes of a node are stored in the chart in the same time, so this copy reduction can prevent exponential explosion of the memory and running-time requirements.

If hidden structure sharing is used, then we have to treat graphs as 'read-only.' Namely, a part of a graph can be shared by some other graph that does not seem to be related, so whenever we change a node in a graph, it has to be copied before the change. Hidden structure sharing cannot occur within a graph, and between graphs that are to be unified, as noted by Malouf *et al.* [20]. The problem is that if this occurs, then we cannot distinguish hidden structure sharing from structure sharing. There are ways to implement hidden structure sharing even in this case (Emele [10]), but they introduce some overhead and significantly complicate the algorithm. If we want to avoid hidden structure sharing in this case, then it is sufficient not to use hidden structure sharing between the grammar (lexicon and rules) and the chart edges (Malouf *et al.* [20]).

In the basic hidden structure sharing approach, a node n can be shared between argument graphs and the result graph only if it is not changed and none of the nodes x such that there is a path $r \to n \to x$ from the root r is not changed. If a node n is changed, then it cannot be shared, but also any of its ancestors, i.e., the nodes between the root and n, cannot be shared. For example, if the node at the end of path g:h was changed in figure 5, then the node at the end of path g could not be shared. Emele [10] extends this hidden structure sharing approach so that only the changed nodes are not shared. In this approach, even though g:h is not shared, the node g can be shared. This is achieved by keeping several versions of each node and by using the global generation counter. Each version of the node is associated with a generation number. At the start of a unification, the global generation number is incremented, and *current environment* is defined as a sequence of valid generation numbers. Using this environment and the global generation number, we can choose for each node the right version, and we can verify whether we are allowed to make changes on the node, or we have to make a copy of it.

This hidden structure sharing method is called *lazy copying.* It has the advantage of eliminating over-copying, early copying, and redundant copying. It eliminates redundant copying in the *strict sense,* i.e., the ancestors of a changed node are not copied. A disadvantage is that there is an overhead cost associated with accessing the right version of the node, similarly to Pereira [22].

Early copying in the strict sense. Tomabechi [26] takes another approach in improving the Wroblewski algorithm. He gives a new definition of early copying, which we will call *early copying in the strict sense.* By this definition, early copying is any node copying done before we know whether the argument graphs can be unified. If they cannot be unified, then no copying is done at all; otherwise, the result graph is copied from the argument graphs. Wroblewski's incremental copying algorithm does not prevent early copying in the strict sense. Tomabechi [26] offers a modified algorithm, called "quasi-destructive graph unification," which prevents early copying in the strict sense.

The problem is solved by having some temporary fields in the original nodes, and instead of creating a node copy, additional changes are stored in these temporary structures. These structures are called *scratch fields.* The nodes that are not changed during unification are not copied. The nodes that are changed are not copied as well, but the scratch fields are used to store new values.

Abstract machine. A different approach to parsing unification-based grammars is taken by Carpenter and Qu [8]. They present an abstract machine for attribute-value logic, with an approach similar to the Warren's Abstract Machine (WAM) [3] for Prolog. This approach is not directly related to our approach, since it does not treat graph unification as a separate issue. Instead, elements of a grammar (such as rules and types) are compiled into the abstract machine code. The code is a complete parser with built-in backtracking mechanism.

The approach is implemented in the system LiLFeS [21, 18, 19].

Van Lohuizen [16, 17] improves the Tomabechi's algorithm by separating the scratch fields into a separate array. This saves some memory and the algorithm becomes thread-safe, i.e., unifications can be done in parallel by different threads on the same collection of graphs. Indexing scratch structures is a problem and two solutions are offered: using a hash-table and using indices associated with nodes.

'Quick-check filtering.' Finally, let us mention the work described by Malouf, Carroll, and Copestake [20], which describes the efficient feature structure operations without compilation. Feature structures are not compiled, as in the abstract-machine approach, but they are used in their original form in unifications. The advantages of not compiling the grammar are: During grammar development, compilation of a large grammar is an expensive operation, which is done frequently; and additionally, the original structure of AVMs is lost due to compilation, which makes grammar debugging difficult.

Malouf *et al.* [20] describe an efficient graph unification, which relies on the Tomabechi's quasi-destructive algorithm. They use hidden structure sharing in non-strict sense, and a new technique called *quick-check filtering.* Quick-check filtering consists of checking for some frequent points of unification failure. These points are identified using statistical methods, and their values are collected in a vector for each graph. Before two graphs are unified, a check on these vectors is done, and if it fails, the unification is not started since we know that it will fail. This method leads to significant running-time savings of 50% [20].

Platform PET. The platform PET [5, 6] is related to the work done by Malouf *et al.* [20]. The platform was designed to be flexible and easy to experiment with different algorithms and approaches. It is a lower-level implementation, which uses the Tomabechi's algorithm with hidden structure sharing, and with improved memory management.

4 Motivation

A frequently used approach to graph unification in the context of NL parsing is the abstract machine proposed by Carpenter and Qu [8]. The method is used, for example, in the system LiLFeS [21, 18, 19]. The graphs are compiled into the code for the abstract machine, and the graph unification is implicitly performed by running the code. As suggested by Malouf *et al.* [20], this approach is not suitable for all applications. Some of the situations where it is not suitable, such as for our parser, include: chart parsers with explicit graph unification operation, which do not need backtracking; during development of large grammars, where compiling is an expensive operation that is done frequently; situations where the loss of the original graph structures is not desirable; and for grammars that do not use *appropriateness* and *well-typedness* of feature structures (Carpenter [7]).

On the other hand, the frequently used graph unification algorithm in the context of NL parsing is Tomabechi's quasi-destructive algorithm [26], which offers a good combination of efficiency and simplicity, especially when enhanced with hidden structure sharing (Malouf *et al.* [20]). Tomabechi algorithm eliminates early copying in the strict sense, i.e., it does not copy any nodes until it is clear that the unification has succeeded and the construction of the result graph is about to start. However, the algorithm uses scratch fields in existing nodes to store intermediate results, which require additional amounts of memory. We observe that *the effect of maintaining scratch fields is essentially equivalent to the effect of node copying.*

The paper that describes the algorithm (Tomabechi [26]) presents some arguments that emphasize practical considerations of expensive garbage collection and dynamic memory allocation:

> *Copying takes time and space essentially because the area in the random access memory needs to be dynamically allocated which is an expensive operation.* (Tomabechi [26])

and

> *This time/space burden of copying is non-trivial when we consider the fact that creation of unnecessary copies will eventually trigger garbage collections more often (in a Lisp environment) which will also slow down the overall performance of the parsing system.* (Tomabechi [26])

Tomabechi's and other algorithms use atomic operations of the intersection of two sets of arcs and of the set complement. If we assume that these operations take constant time, then each algorithm is linear in the graph size. However, they are not constant: a straightforward implementation leads to a linear running time of those operations, which means that the unification algorithms are quadratic.

The unification algorithm can be significantly simplified and optimized by rephrasing it at a lower level with direct and efficient memory management, without hidden details, and even without recursion or function calls.

One of the problems with previous algorithms that use the global generation counter is that when automatic garbage collection is used the garbage collector cannot free the objects that are invalidated only by incrementing the global counter.

Although some newer contributions ([16, 17, 5, 6]) address the issue of a better memory management scheme, they still use the old algorithm framework with some old drawbacks. For example, the memory scheme can be further simplified and optimized; instead of using arc lists we can use arrays of arcs. It is not clear that the previous algorithms avoid memory fragmentation, which leads to inefficient running-time in garbage collection as well as in dynamic memory allocation.

This approach. After taking all this into account, we provide a model which addresses these issues:

- It is a simple model, and it explicitly solves the problem of graph unification in the context of chart parsing.

- It does not use compilation, does not change the original graphs, and does not require well-typedness and appropriateness, while providing the way to add them.

- It uses the 'global counter trick', hidden structure sharing, and it handles cyclic graphs.
- The algorithm is simple and complete, without any complex atomic operations. It is efficient and flat (does not include recursive calls or any function calling). One of the advantages of not expressing arc-set operations (complement and intersection) as atomic operations is that we use sub-node hidden structure sharing, i.e., structure sharing at the edge level.
- The memory model is simple: consists of an array and a small group of variables. There is no use of garbage collection or memory allocation system calls, unless we want to expand the array. The memory is not fragmented.
- The model is a low-level machine which can be directly translated to a low-level language like C.

We do not want to imply that recursive functions and function calls are necessarily a disadvantage. It is usually not a difficult task to turn a recursive function into a non-recursive one, and modern compilers provide *inline* functions, which avoid inefficient function calls. But, by expanding the algorithm completely as we did, and by optimizing it in this way, we also achieve a new higher level of understanding the algorithm, and its efficiency is not dependent on some external mechanisms, which are not necessarily guaranteed.

5 Graph Unification Machine

Before describing the machine in detail—the memory model and the algorithm—let us first give a higher-level overview.

The memory model consists of an array of integer *cells* and some integer variables. A graph edge is represented as a pair of cells: an attribute and the address of the end node. An attribute is represented as a negative integer smaller or equal than -2. The term *address* refers to the index of the initial cell of a node. The edges coming out from the same node are grouped into sequences, which are sorted by the attribute numbers in ascending order. For example, to sort edges in alphabetic order, we use the following encoding for attributes:

attribute:	a	b	d	e	g	h
id number:	-7	-6	-5	-4	-3	-2

The number '-1' is reserved as the sentinel that marks the end of the sequence. For example,

0	1	2	3	4
-7	8	-5	18	-1

denotes a node with two edges: one with the label a (-7), which leads to the node at address 8 (array index 8), and the other with the label d (-5), which leads to the node at address 18. Instead of writing actual attribute codes, we can replace them with labels for the purpose of clearer presentation:

0	1	2	3	4
a	8	d	18	-1

Small indices above the cells are used to denote the absolute position within the array.

The type of a node is encoded in another cell, which we call the T cell. We do not present details about unification of typed feature structures, but we do discuss how the type identifiers stored in T cells can be easily used to add type unification to presented algorithm. We also discuss how the algorithm can be adapted to implement appropriateness and well-typedness. A type is simply encoded as an integer, and in actual implementations it can be looked up in a table. Here, we only assume that $T = 0$ for a *leaf* type, and $T \neq 1$ for a *complex* type. As in Pereira [22], we define: a *leaf* to be a place holder used in structure sharing, and it can be unified with an atom or a complex type; a *complex* node is a node that can have outgoing edges and it cannot be unified with atoms; and, an *atom* cannot have any outgoing edges, and it can be unified only with a leaf or the same atom.

When a node is changed in a non-destructive way, we use a generation counter and a forward pointer using two cells: G and F. These cells are used only during unification: if G is equal to the current *Generation* and $F \geq 0$, then F is a forward pointer to the actual copy of the node.

Finally, we can describe a typical node representation: It consists of the cells G, F, and T, and a sequence of outgoing edges. For example,

0	1	2	3	4	5	6	7
1	112	1	a	8	d	18	-1
G	F	T					

(2)

denotes a complex node ($T = 1$), with two outgoing edges. The cells G and F can be ignored between unifications. If $T = 0$, the node is a leaf node, which implies that it does not have any outgoing edges, hence there is no need for the '-1' sentinel. For example,

0	1	2	3	4	5	6
0	0	1	-1	0	0	0
G	F	T		G	F	T

represent two nodes. The first one (at address 0) is a complex node without outgoing edges (it cannot be unified with an atom), and a leaf node (at address 4).

During unification, whenever a node is accessed we first check if $G = Generation$. If this is true and $F \geq 0$ then we have to follow the address contained in F (forward) to find the actual value of the node. Actually, it is normally sufficient just to check $G = Generation$, but there is a special case where $G = Generation$ and $F < 0$, which will be discussed later. This procedure is called *dereferencing*. The new node may also contain a forward reference to another node, and so on. The sequence of nodes visited in this way is called a *reference chain*. *Path compression* is performed in each dereference for efficiency reasons; i.e., the F cells of all nodes except the last two in the reference chain are updated with the address of the last node.

An atom cannot be changed, so there is never a need to make forward reference from an atom; i.e., there is no need for G and F cells in this case. In order to disambiguate between atoms and other nodes, we assume that the value of a G cell is always non-negative, and we encode atoms with negative numbers. Hence an atom node occupies just one cell, with its negative integer id value. Similarly to attributes, we will present an atom cell as $\boxed{\text{A}}$ instead using the actual stored value, i.e., $\boxed{-1}$, if the atom 'A' is encoded as -1.

A node does not necessarily occupy a continuous memory location. It can be fragmented into a linked list of memory locations during unification, and permanently due to *sub-node hidden structure sharing.* This is done by using non-negative integers in cells normally occupied by attributes or the '-1' sentinel. Whenever such a number is encountered, we dereference it and continue to read sequence at the given memory location. For example, the node 2 can have the following fragmented representation in the memory:

0	1	2	3	10	11	12	20	21	22
1	112	1	20	d	18	-1	a	8	10
G	F	T							

(3)

Our algorithm relies on the Wroblewski and Tomabechi algorithms, with hidden structure sharing addition (similarly to [20, 5, 6]). However, the distinction between Wroblewski and Tomabechi algorithms is greatly reduced in the context of our memory model. One of the points of this work is to show that there is no essential difference between copying the changed node and not copying it but using a scratch structure to keep a temporary list of edges.

The algorithm proceeds in two phases: In the first phase, we unify the graphs, similarly to Wroblewski's and Tomabechi's `unify1` and `unify-dg` algorithms. Recursion or any kind of function calling is avoided, since it introduces processing overhead. If a node is not changed, even after it is unified with another node, it is hidden structure shared. Parts of a node, i.e., edges, can also be structure shared. Instead of recursive calls, we use a stack to store addresses of node pairs to be unified. The stack is also used in path compression. If unification fails, we simply invalidate all forward cells for future unifications by increasing the generation counter by 1.

If unification succeeds, the second phase is executed, which is *copying.* The result graph is consolidated, by copying and dereferencing. This is also done without recursive calls. The depth-first search is performed using the stack. The high-level pseudo-code for the first phase is given in algorithm 1, and for the second phase is given in algorithm 2.

5.1 Memory model

The memory model can be divided into two parts: the static part, which keeps its state between unifications, and the nonstatic part, whose value can be discarded between unifications. We adopt the convention of using capitalized names for the static variables. All variables are integer variables, and all arrays are arrays of integers.

Static memory. The static part of the memory model consists of the array A, having the length *A_len*, the index *Alloc_last*, which denotes the last used cell of the array, and the generation counter *Generation*. The initial value of *Alloc_last* is -1, and the initial value of *Generation* is 1. Figure 6 depicts the usage of the array A. We will also refer to the array A as the *memory.* The first part of the array contains all graphs in the passive and active charts, from index 0 to index *Alloc_last*. This is the useful part of the

Algorithm 1 First phase

```
initialize memory and verify the amount of free memory
push addresses of argument graphs on stack
while stack not empty do {First phase}
    pop two node addresses and dereference them
    if the nodes are equal then
        continue
    if at least one node is leaf then
        make forward to another node
    else if at least one node is atom then
        unification fails, return
    else {both nodes are complex nodes}
        merge arc lists into a new location, with pushing node addresses
        to be unified later on stack, and sharing a tail of the list if possible
        if the result can be embedded in one node then
            embed the result in the node
        else if existing location can be used then
            make forwards to existing location
        else
            repackage the result into existing node(s)
```

array between unifications, i.e., the static part. The initial value of G cells for all graphs in the chart is 0.

Additionally, we need to keep initial addresses (array indices) of all graphs that the parser maintains, and their sizes. This is static information, but we do not include it in our memory model, since it is application dependent and should be maintained externally.

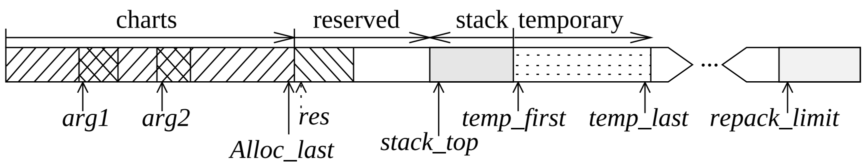

Figure 6: Array usage

Algorithm 2 Second phase

```
initialization; state ← FORWARD; i ← root (i is the current node)
while state ≠ END do
    if state = FORWARD then
        dereference i
        if i is atom, or it was visited, or it is being visited,
                or it does not have edges and it is in charts area then
            state ← BACKWARD, update size, and mark i as visited (F ← −1)
        else
            push copy of i to reserved area, with replacing F with negative
            address of F, and attributes with addresses of original attributes
            if i has no edges then
                if i is in charts area then
                    update size and pop copy from reserved area
                else i ← copy of i
                mark i as visited, state ← BACKWARD
            else
                push the pointer address of the last child, and set i to the pointer
    if state = BACKWARD then
        if stack is empty then state ← END
        else
            pop the edge pointer address from stack
            if the previously visited tail is shared, and this edge can be shared
                    then
                increase size and share this edge
            else
                update pointer with i
                replace attribute address with attribute
            if there are more edges then
                push new pointer address on stack
                i ← the pointer; state ← FORWARD
            else
                if the whole node can be shared then
                    pop it from reserved area, update size, and set i
                else i ← the node address
add the size of newly occupied memory to size; increment Generation
return i and size
```

Nonstatic memory. In order to unify two graphs, we need to know their initial addresses *arg1* and *arg2*, and their sizes *size1* and *size2*. Figure 6 may be misleading since it shows two graphs occupying continuous portions of the memory. They can be fragmented in general, due to hidden structure sharing. The size of a graph is the number of cells used by the graph, excluding the reference cells in fragmented nodes due to sub-node sharing.[4]

Before starting unification, some memory space starting from $Alloc\_last + 1$ is reserved for the result graph. This can be done since we can calculate an upper bound on the size of the result graph using the sizes of the argument graphs.

Additionally, the unification procedure makes use of a stack. The stack can be handled as a distinct array, but we find it convenient to use a part of the array A as the stack. We can calculate an upper bound on the size of stack, so this possible. The *reserved* and *stack* areas span from the index $Alloc\_last + 1$ to $temp\_first - 1$.

Indices *temp_first* and *temp_last* delimit the temporary scratch area used during unification, which grows by increasing the index *temp_last*. If unification succeeds, the result graph is copied at the position $Alloc\_last + 1$, and the index *Alloc_last* is updated with the new index of the last used cell. The size of the result graph is determined in the process. Parts used in hidden structure sharing are not copied.

We will see that we can save some time if we do not care about fragmentation of the temporary space during unification. However, in that case the temporary area can grow up to the quadratic size to the size of the argument graphs. If we do care about fragmentation, we apply a procedure called *repackaging* in some cases. The procedure takes some extra running time, but it can keep the size of the temporary area linear to the size of the argument graphs. In order to do this efficiently, we calculate an index called *repack_limit*. If *temp_last* becomes equal or greater than *repack_limit*, we apply repackaging in order to finish unification within available space. Otherwise, we can use non-repackaging method, without worrying about the large temporary area.

This memory model provides for a simple and efficient memory management. We do not have to handle memory fragmentation since the chart always occupies a continuous initial portion of the array. If unification fails,

[4]The total size of several graphs can be larger that the actual number of cells they occupy due to hidden structure sharing.

there is no need for any memory clean-up. We simply increase *Generation* counter and stop execution. There is no need for status fields, as in previous algorithms: indices provide an easy way to determine a node status. If index is less or equal *Alloc_last* the node is not copied, if it is greater or equal *temp_first* it is a temporary copy, or it is a final copy otherwise.

Upper bounds on sizes. As already mentioned in this subsection, we can calculate upper bounds on the sizes of the reserved and stack areas, and we calculate the repackaging limit. These values are calculated at the beginning of each unification, and we verify the size of the memory area. Hence, there is no need to do array-bounds checking afterwards, during unification, which improves the running time of the algorithm.

The size of the reserved area is determined by the upper bound on the size of the result graph: *max_size*. At the beginning of the graph unification, we start with two argument graphs with the total size $\mathit{size1} + \mathit{size2}$. During unification, these graphs are unified in a pseudo-destructive way, using their copies in the temporary area. We will refer to the size of this temporary graph as the *temporary graph size.* In the *temporary graph size* we do not count forward cells; i.e., G and F cells of the forward references, and references in fragmented edge sequences. In other words, temporary graph size includes only G-F-T cells of final nodes, 2 cells per edge, and '-1' sentinels.

Whenever two nodes are unified, the temporary graph size decreases, and it also decreases due to edge unification. If at least one argument graph is atom, the size of the result graph is at most 1; otherwise, it is the total size of the argument graphs decreased by at least 3. Hence, an upper bound on the size of the result graph, i.e., the size of the reserved area is:

$$\mathit{max\_size} = \max(1, \mathit{size1} + \mathit{size2} - 3). \tag{4}$$

In order to find an upper bound of the stack size, we consider two phases of the algorithm:

In the first phase, the stack is used to store addresses of node pairs that need to be unified, and it can temporarily store an array of addresses of F cells for path compression during dereferencing. If at least one argument graph is an atom, then a stack of size 2 is sufficient. We further assume that the argument graphs are not atoms. Each pair of node addresses stored on stack to be unified later is associated with a future node unification. Each link in a reference chain is associated with one finished node unification.

Since the initial root unification reduces the temporary graph size by at least 3 (argument graphs are not atoms), and any other node unification reduces the size by at least 3 (at least 1 for nodes, and 2 for the associated edge unification), the total number of node unifications is at most $\frac{1}{3}(\mathit{size1} + \mathit{size2})$ (because the size of the temporary graph cannot be negative). Hence, during the first phase, we do not need more than

$$\max(2, \frac{2}{3}(\mathit{size1} + \mathit{size2}))$$

cells on stack (2 cells are used at most for one node unification).

In the second phase, the stack is used to keep addresses of the currently visited complex nodes, and the currently visited edges in a depth-first search of the result graph, and temporarily for path compression during dereferencing. As we saw before, each reference in a reference chain is associated with a reduction of at least 3 of the temporary graph size. A currently visited complex node occupies at least 6 cells of the final graph size (G-F-T, '-1' sentinel, and 2 cells for at least one edge). We conclude that during the second phase we do not need more than

$$(\mathit{size1} + \mathit{size2})/3$$

cells on the stack.

Hence, it is sufficient to reserve the stack area of

$$\max(2, \left\lfloor \frac{2}{3}(\mathit{size1} + \mathit{size2}) \right\rfloor) \tag{5}$$

cells.

Regarding the temporary area, we first note that it is possible to maintain temporary graph structure in $\frac{3}{2}\mathit{max\_size}$ cells. Namely, during unification some edge lists can be defragmented, so we may need 3 cells instead of 2 for some edges. The factor $\frac{3}{2}$ is used to accommodate these extra cells. Additionally, for a short-term merge operation, we may need additional *max_size* cells. Hence, we may need $2.5 \cdot \mathit{max\_size}$ cells in total, so we use the parameter *repack_limit* to mark the last $2.5 \cdot \mathit{max\_size}$ cells, which is later used as the 'repackaging trigger limit':

$$\mathit{repack\_limit} = \mathit{A\_len} - \lfloor 2.5 \cdot \mathit{max\_size} \rfloor \tag{6}$$

This limit is used in two instances: first, if $\mathit{temp\_first} \leq \mathit{repack\_limit}$ initially, then we are guaranteed to have sufficient memory for unification; second, if $\mathit{temp\_last} \geq \mathit{repack\_limit}$, we have to apply repackaging.

5.2 Algorithm

We present the unification method as a single, flat algorithm, and consequently it is relatively long. For this reason, we do not present it separately and then describe it, but the comments are interleaved with the algorithm specification.

The algorithm pre-conditions and post-conditions are the following:

Require: *arg1* and *arg2* are starting addresses of the argument graphs, and *size1* and *size2* are their sizes

Ensure: -1 is returned if unification fails, and the starting address *res* of the result graph and its size *size* is returned otherwise. If the procedure fails due to insufficient memory, the special token ERROR is returned.

In an actual parser, it is usually preferable to attempt to increase the array size, than to return 'ERROR'.

We first calculate parameters and initialize some variables according to the discussion in the last section (equations (4,5,6)):

$\mathit{max\_size} \leftarrow \max(2, \mathit{size1} + \mathit{size2} - 3)$

$\mathit{temp\_first} \leftarrow \mathit{Alloc\_last} + 1 + \mathit{max\_size} + \lfloor \max(2, \frac{2}{3}(\mathit{size1} + \mathit{size2})\rfloor$

$\mathit{temp\_last} \leftarrow \mathit{temp\_first} - 1$

$\mathit{stack\_top} \leftarrow \mathit{temp\_first}$

$\mathit{repack\_limit} \leftarrow \mathit{A\_len} - \lfloor 2.5 \cdot \mathit{max\_size} \rfloor$

We verify that the memory size is sufficient:

---

if $\mathit{temp\_last} \geq \mathit{repack\_limit}$ **then return** ERROR

The stack is initialized:

---

push *arg1* on *stack*

push *arg2* on *stack*

5.2.1 First phase

The first phase is unification of the graphs using the temporary memory space and forward references. If unification fails, then execution stops and -1 is returned. Otherwise, the second phase continues. The first phase is a loop, which iterates until the stack is empty:

---

while $\mathit{stack\_top} < \mathit{temp\_first}$ **do**

    $i_2 \leftarrow$ pop stack; $i_1 \leftarrow$ pop stack

    if $i_1 = i_2$ **then continue**

Variables i_1 and i_2 are used as indices to the two argument graphs. The keyword '**continue**' starts a new iteration of the loop.

Dereferencing. Variable i_1 is dereferenced with path compression. Dereferencing is finding the actual version of the node, and path compression is updating traversed nodes to the address of the actual node, so that later dereferences are faster. The technique is used in the UNION-FIND structure for representing disjoint sets (Cormen *et al.* [9]), and when combined with ranking it gives practically constant amortized time[5] for dereferencing, i.e., for the FIND operation. Ranking can be easily applied here if we add an additional cell to the G-F-T cells in each node, but it would use additional memory and practical time-saving significance is questionable. Asymptotically, it is not significant since the merge operation that follows takes linear time in the worst case anyway.

If i_1 needs dereferencing, then we follow the forward chain of references until the node is found while storing the addresses of the traversed F cells on stack, and we update the F cells afterwards. There is no need to update the F cells of the last two nodes in the chain. The number '-1' is used as the sentinel on the stack.

```
12: if A[i1] = Generation then
13:     j ← A[i1 + 1]                                        {F cell}
14:     if A[j] = Generation then
15:         push −1 on stack
16:         repeat
17:             push i1 + 1 on stack
18:             i1 ← j; j ← A[j + 1]
19:         until A[j] < Generation
20:         while A[stack_top] > −1 do A[pop stack] ← j
21:         pop stack
22:     i1 ← j
```

Variable j is a temporary variable with the scope limited to the above part of the algorithm.

Variable i_2 is dereferenced with path compression in the same way (it is sufficient to replace i_1 with i_2 in lines 12–22:

[5] $O(\alpha(n))$ time, where α is the inverse of the Ackermann's function. $\alpha(n) \leq 5$ for all practical purposes.

```
dereference i2 with path compression
if i1 = i2 then continue
```

Unification with an atom or a leaf. At this point we have two different nodes that have to be unified. First we handle the case where at least one node is a leaf node:

```
if A[i1] ≥ 0 and A[i1 + 2] = 0 then {i1 is leaf}
    A[i1] ← Generation; A[i1 + 1] ← i2                {forward to i2}
else if A[i2] ≥ 0 and A[i2 + 2] = 0 then
    A[i2] ← Generation; A[i2 + 1] ← i1
```

Otherwise, we check for the case where at least one node is an atom:

```
else if A[i1] < 0 or A[i2] < 0 then
    if A[i1] ≠ A[i2] then
        Generation ← Generation + 1; return −1
```

As we can see, if unification fails, then we just increment *Generation* by 1 and return -1.

Unifying two complex nodes. In the last case, we know that both nodes are complex nodes, and we proceed with merging their edge lists. If the nodes were typed, we would first unify the types. The lists are sorted, so they are merged in linear time into a new list located after the end of the temporary area. If an attribute appears in both lists, the corresponding pair of addresses is pushed on the stack for later unification. If we used well-typed feature structures, we would also push on the stack the id number of the appropriate type. We discuss later in detail how this can be done.

During merging we maintain two variables $embed_1$ and $embed_2$ to keep track of embedding information: $embed_k = 0$ means that the resulting node can be embedded into node i_k so far, and $embed_k > 0$ means that the tail of the resulting node starting from index $embed_k$ can be shared with node i_k starting from index src_k. After merging, $embed_k = -1$ is used to denote that no tail can be shared between the resulting node and the node i_k. Indices j_1 and j_2 are used during merging to read input sorted lists, and j_3 is used to produce the output.

```
else
    embed₁ ← 0; embed₂ ← 0; src₁ ← i₁; src₂ ← i₂
    j₁ ← i₁ + 3; j₂ ← i₂ + 3; j₃ ← temp_last + 4
    A[j₃ − 1] ← 1                                  {set F of result}
    while A[j₁] ≠ −1 and A[j₂] ≠ −1 do
        while A[j₁] ≥ 0 do j₁ ← A[j₁]
        while A[j₂] ≥ 0 do j₂ ← A[j₂]
        if A[j₁] < A[j₂] then
            A[j₃] ← A[j₁]; j₁ ← j₁ + 1; j₃ ← j₃ + 1
            A[j₃] ← A[j₁]; j₁ ← j₁ + 1; j₃ ← j₃ + 1
            embed₂ ← j₃; src₂ ← j₂
        else if A[j₂] < A[j₁] then
            A[j₃] ← A[j₂]; j₂ ← j₂ + 1; j₃ ← j₃ + 1
            A[j₃] ← A[j₂]; j₂ ← j₂ + 1; j₃ ← j₃ + 1
            embed₁ ← j₃; src₁ ← j₁
        else
            A[j₃] ← A[j₁]
            j₁ ← j₁ + 1 j₂ ← j₂ + 1; j₃ ← j₃ + 1
            if A[j₁] ≠ A[j₂] then
                push A[j₁] on stack; push A[j₂] on stack
            A[j₃] ← A[j₁]
            j₁ ← j₁ + 1 j₂ ← j₂ + 1; j₃ ← j₃ + 1
    if A[j₁] ≠ −1 then embed₂ ← −1
    else if A[j₂] ≠ −1 then embed₁ ← −1
```

Location of the result node. After merging lists of edges the resulting list is located at the position $temp\_last + 4$, so that after updating G-F-T cells at $temp\_last + 1$ the new node can be appended to the temporary area. There are four possible outcomes regarding the position of the resulting node:

1. it can be embedded at position i_1, and we ignore the merged list,
2. it can be embedded at position i_2,
3. it is added to the temporary area without copying the edge list, or
4. the resulting edge list has to be repacked due to memory shortage using the space used by the argument nodes at i_1 and i_2, provided that at

least one of them is already located in the temporary area.

The first two cases are handled in the following way:

```
        if embed_1 = 0 and (embed_2 ≠ 0 or i_1 ≤ i_2) then
            A[i_2] ← Generation; A[i_2 + 1] ← i_1
        else if embed_2 = 0 then
            A[i_1] ← Generation; A[i_1 + 1] ← i_2
```

Otherwise, in the next two cases we first decide about tail sharing. Usually, a tail can be shared only with one node, but if it can be shared with both nodes, we choose the tail ending in the charts area. If the question still cannot be resolved, we choose the longer tail.

```
        else
            if embed_1 > −1 and (embed_2 = −1 or (j_1 ≤ Alloc_last and
                    j_2 > Alloc_last) or embed_1 ≤ embed_2 then
                j_3 ← embed_1; A[j_3] ← src_1; embed_2 ← −1
            else j_3 ← embed_2; A[j_3] ← src_2; embed_1 ← −1
```

The first of the two remaining cases is to add the node to the temporary area. This is done if we have sufficient memory, or if nodes i_1 and i_2 cannot be used for repackaging.

```
            if j_3 < repack_limit or
                    (i_1 < temp_first and i_2 < temp_first) then
                A[temp_last + 1] ← 0
                A[i_1] ← Generation; A[i_2] ← Generation
                A[i_1 + 1] ← temp_last + 1; A[i_2 + 1] ← temp_last + 1
                temp_last ← j_3
```

Otherwise, we must repackage the resulting node into the space occupied by the argument nodes, or at least we reuse one node if the other node is in the charts area. First, we may have to swap i_1 and i_2, so that we can write at the position of i_1. If possible, we also prefer i_1 not to be the node which shares a tail with the result:

```
            else
                if i_1 < temp_first or
                        (embed_1 > −1 and i_2 ≥ temp_first) then
                    swap i_1 and i_2
                A[i_2] ← Generation; A[i_2 + 1] ← i_1
```

```
         A[i1] ← 0
```

Copy T cell, prepare indices i_1, i_2, and j_1, which will be the source index for copying, and start the loop. If we reach the end of the reusable edge sequence, we look for the next available space (i_2 or extend temporary area):

```
         A[i1 + 2] ← A[temp_last + 3]
         j1 ← temp_last + 4; i1 ← i1 + 3; i2 ← i2 + 3
         while A[j1] < −1 do
             if i1 < temp_last and A[i1] ≥ −1 then
                 ii ← i1
                 while A[ii] ≥ 0 do ii ← A[ii]
                 if ii < temp_first then
                     if i2 ≥ temp_first then ii ← i2; i2 ← −1
                     else ii ← temp_last + 1
                 A[i1] ← ii; i1 ← ii
             if i1 = A[j3] then {don't overwrite shared tail}
                 ii ← i1
                 while A[ii] < −1 do
                     A[j3] ← A[ii]; A[j3 + 1] ← A[ii + 1]
                     j3 ← j3 + 2; ii ← ii + 2
                 while A[ii] > 0 do ii ← A[ii]
                 if A[ii] = −1 then A[j3] ← −1
                 else A[j3] ← ii
             A[i1] ← A[j1]; A[i1 + 1] ← A[j1 + 1]
             i1 ← i1 + 1; j1 ← j1 + 1
         A[i1] ← A[j1]
         if i1 > temp_last then temp_last ← i1
```

5.2.2 Second phase

If the unification was successful in the first phase, the new graph is copied to the reserved area. During copying, all references are resolved to direct node addresses, and all nodes are defragmented except for tail sharing with the nodes from charts area. Whenever possible, hidden structure sharing is used with previous nodes, but if we do not want some nodes to be shared in this way, we can easily prevent it.

The algorithm makes a depth-first search through the new graph. The status of a node is determined in the following way:

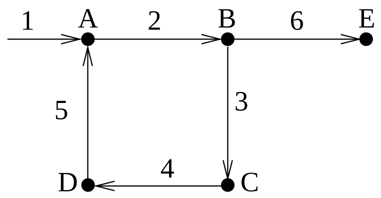

Figure 7: The shared cycle problem

$G <$ *Generation*	:	node is not visited
$G =$ *Generation* and $F \geq 0$	:	node is part of a reference chain
$G =$ *Generation* and $F < 0$	:	node visited, or being visited, its address is final.

As in the first-phase algorithm, we avoid using recursion or function calling, but use the stack instead. Whenever an unvisited node is found, it is copied to the reserved area with the value of F cell being equal negative address of the original F cell (it cannot be 0), and with replacing all attributes with addresses of the original attributes. All edges are visited starting from the last one. In this way, we can easily determine the maximal tail that can be shared with the charts area, or if the whole node can be shared.

Cycles. An interesting issue is handling of cycles, since the algorithm does allow cyclic graphs. The problem is illustrated in figure 7. The edges in figure 7 are numerated in order in which they are visited. Let us assume that nodes A, B, C, and D are not changed during the first phase. The problem is whether the node D should be copied. If node E, which has not been visited yet, is changed, we should copy all the nodes; otherwise, if the node E is shared, all other nodes in figure should be shared as well.

One solution is to repeat the depth-first search until such anomalies are resolved. However, this can lead to a quadratic search instead of a linear. For example, let us assume that in the graph in figure 8 only the node G has been changed. After the first visit, we detect that node A should have been copied. It can be copied in the next visit, as well as nodes B and C, but the nodes E and F are not copied, so one more depth-first visit is required. If we extend example as shown in figure 9, then we need $O(n)$ depth-first visits.

Another solution is not to allow hidden structure sharing of cycles, i.e., we always copy them. We use this approach because it is faster and it seems that cycles do not appear that frequently in practical NL grammars. For

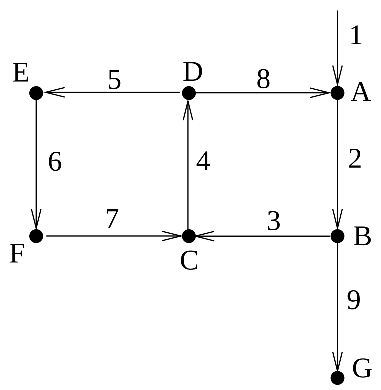

Figure 8: Two shared cycles

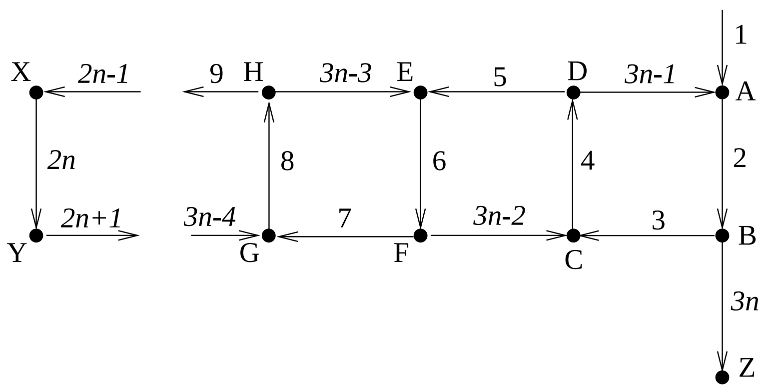

Figure 9: $O(n)$ shared cycles

example, the well-known LINGO grammar is an acyclic grammar.

The algorithm consists of a loop and we distinguish three states using variable *state*: 'FORWARD' for descent to child nodes, 'BACKWARD' for backing up, and 'END' to denote end of the second phase. Instead of updating *Alloc_last*, we use *new_Alloc_last* and update *Alloc_last* at the end. In variable *size* we accumulate the size of the hidden-structure-shared part, and the size of the non-shared part ($new\_Alloc\_last - Alloc\_last$) is added at the end. Variable i contains the address of the currently visited node. First, the variables are initialized:

```
size ← 0; new_Alloc_last ← Alloc_last
i ← arg1; state ← FORWARD
```

We consider the FORWARD case. Variable i contains the address of the node: it is dereferenced in a similar way as in the first phase.

```
while state ≠ END do
    if state = FORWARD then
        if A[i] = Generation and A[i + 1] ≥ 0 then
            j ← A[i + 1]
            if A[j] = Generation and A[j + 1] ≥ 0 then
                push −1 on stack
                repeat
                    push i + 1 on stack
                    i ← j; j ← A[j + 1]
                until A[j] ≠ Generation or A[j + 1] < 0
                while A[stack_top] > −1 do
                    A[pop stack] ← j
                pop stack
            i ← j
```

If the node is an atom, or it was visited, then we go backward. Variable i contains the address of the node.

```
        if A[i] < 0 then
            state ← BACKWARD; size ← size + 1
        else if A[i] = Generation and A[i + 1] ≤ −1 then
            state ← BACKWARD
        else if i ≤ Alloc_last and (A[i + 2] = 0 or A[i + 3] = −1) then
            A[i] ← Generation; A[i + 1] ← −1
```

```
      if A[i + 2] = 0 then size ← size + 3
      else size ← size + 4
```

Otherwise, the node is first copied to the reserved area, with replacing the value of F cell with the negative address of the original F cell, and with replacing attributes with addresses of the original attributes.

```
    else
      A[i] ← Generation; A[i + 1] ← new_Alloc_last + 1
      A[new_Alloc_last + 1] ← Generation
      A[new_Alloc_last + 2] ← −i − 1                              {F cell}
      A[new_Alloc_last + 3] ← A[i + 2]
      new_Alloc_last ← new_Alloc_last + 3
      src ← i + 3
      loop
        while A[src] ≥ 0 do src ← A[src]
        if A[src] = −1 then
          new_Alloc_last ← new_Alloc_last + 1
          A[new_Alloc_last] ← −1
          break
        A[new_Alloc_last + 1] ← src
        A[new_Alloc_last + 2] ← A[src + 1]
        new_Alloc_last ← new_Alloc_last + 2 src ← src + 2
      i ← A[i + 1]
```

If the node has no edges, then we go backward. Otherwise, we start visiting edges starting from the last one.

```
      if A[i + 3] = −1 then
        if −A[i + 1] ≤ Alloc_last then
          A[−A[i + 1]] ← −1
          new_Alloc_last ← new_Alloc_last − 4
          size ← size + 4
          i ← −A[i + 1] − 1
        state ← BACKWARD
      else
        push new_Alloc_last − 1 on stack
        i ← A[varnew_Alloc_last − 1]
```

The FORWARD state is finished. In the BACKWARD state, we first check if the depth-first search is finished. If it is not finished, we check whether the finished edge can be tail-shared. If it cannot, we update the address of the destination node.

```
if state = BACKWARD then
    if stack_top = temp_first then state ← END
    else
        p_adr ← pop stack
        if A[p_adr + 1] ≥ −1 and i ≤ Alloc_last and
                i = A[p_adr] and A[p_adr − 1] ≤ Alloc_last then
            new_Alloc_last ← new_Alloc_last − 2
            size ← size + 2
        else
            A[p_adr] ← i
            A[p_adr − 1] ← A[A[p_adr − 1]]
        p_adr ← p_adr − 2
```

We visit the next edge. If there are no more edges, we check if the whole node can be shared, and go backward:

```
        if A[p_adr − 1] ≥ 0 then
            push p_adr on stack
            i ← A[p_adr]; state ← FORWARD
        else
            if A[p_adr + 1] ≥ 0 and − A[p_adr − 1] ≤ Alloc_last then
                new_Alloc_last ← new_Alloc_last − 4
                size ← size + 4
                i ← −A[p_adr − 1] − 1
                A[i + 1] ← −1
            else i ← p_adr − 2
```

At the end, we update the size and *Alloc_last*, and increment *Generation*.

```
size ← size + new_Alloc_last − Alloc_last
Alloc_last ← new_Alloc_last
Generation ← Generation + 1
return (i, size)
```

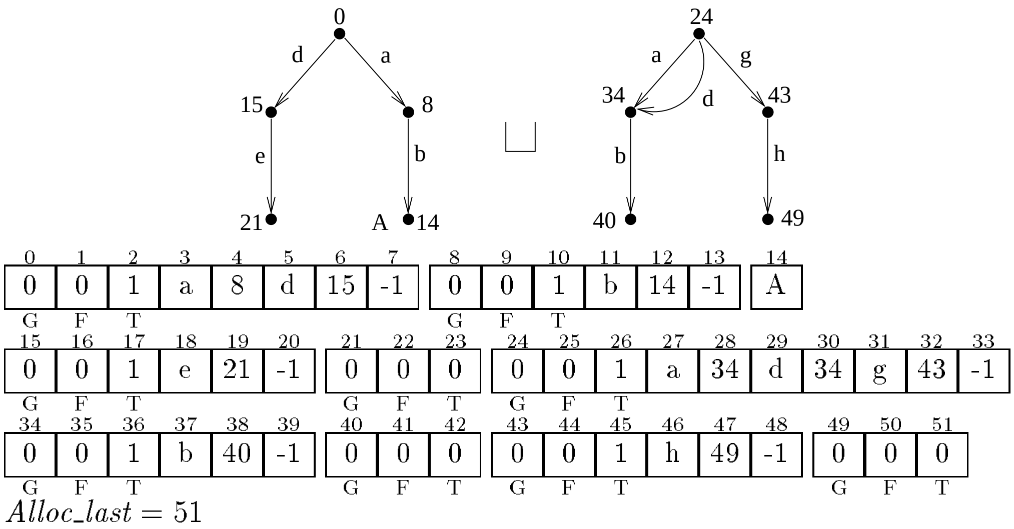

Figure 10: Graph encoding

6 Example

In this section we illustrate the memory layout and the algorithm on an example. Figure 10 illustrates how the two graphs used previously in section 3 are encoded. The following attribute encoding is used:

$$\begin{array}{cccccc} a & b & d & e & g & h \\ -7 & -6 & -5 & -4 & -3 & -2 \end{array}$$

The root of the first graph is located at position 0, and the graph has size 24. Unlike the graphs in section 3, the node at the path a:b is atom, which is introduced to illustrate the atom representation as well. The second graph is located at position 24, and its size is 28. The last allocated cell is 51. The graphs shown in the figure includes the starting addresses of all nodes. The atom A is represented as negative number -1.

After executing the first phase of the unification algorithm, the layout shown in figure 11 is obtained. The unification has succeeded, and the resulting graph can be found starting from address 0. In this intermediate stage, forward references are used, so the graph shown includes the node addresses obtained after dereferencing. The temporary area starts at position 136 and ends at 141.

Figure 12 shows the memory layout after the second phase of the algo-

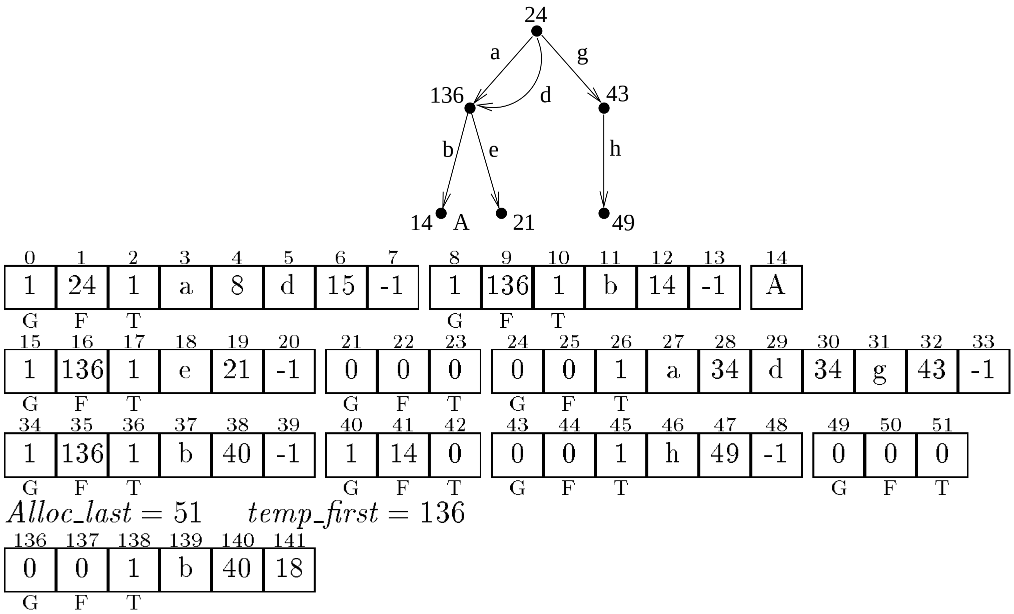

Figure 11: Memory after the first phase

rithm. The shaded area of the result graph covers the shared nodes and edges. The result graph starts at address 52 and its size is 31.

7 Discussion

When is hidden structure sharing allowed? We have already discussed in section 3 the problem that hidden structure sharing cannot occur within a graph, and between graphs that are to be unified (as noted by Malouf *et al.* [20]). In our approach, the problem can be solved by storing all encoded grammar rules in the first part of the memory. This *rules area* occupies a continuous initial part of the memory, so we can easily tell if a node belongs to an original rule image by comparing the node address with the last occupied address of the rules area. Whenever this is the case, the node cannot be shared.

One way to handle lexicon entries is to store them in the rules area. Because of the lexicon size, a better approach is to add a new copy of a lexical entry during char initialization from some external source. The original addresses used in encoding of a lexical entry can be relative, and they are

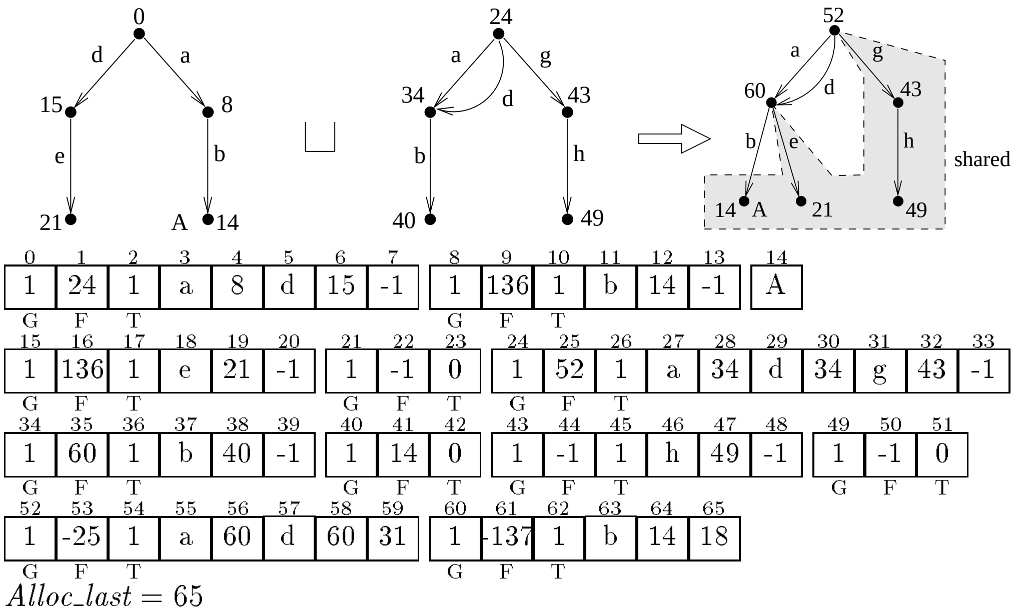

Figure 12: Memory after the second phase

recalculated according to the final absolute position of the entry.

Another way to handle the problem is to allow hidden structure sharing in all situations, but to introduce the notion of *safe* hidden structure sharing which could be safely used within one graph or between argument graphs. This approach would be similar to the solution given by Emele [10]. However, this would add significant complexity to the original algorithm and it is not clear without empirical evaluation that it would lead to any kind of performance improvement with actual NL grammars.

Typed feature structures. As in the Wroblewski's and Tomabechi's work, we do not include any feature-structure types, except distinguishing between an atom, leaf, and complex node. However, the T cell in the node reserves the whole integer range (without 0), for type identifiers. When two nodes are unified their types could be unified by calling an external procedure, and the resulting type would be stored in the T cell of the resulting node. Hence, our algorithm can be easily adapted for handling typed feature structures.

In order to support well-typedness and appropriateness, we have to make sure that in merging edge lists all destination nodes have appropriate types. This can be solved by going through the list of appropriate attributes of the type and by pushing a pair of a type and a node address to the stack for each attribute that appears in the merged list. These type restrictions would be resolved later, after popping the information from the stack.

Discontinuous memory. Although our continuous memory model provides a simple and efficient solution, it can be desirable in practical applications to have a discontinuous memory, with stack, temporary area, and parts of the charts area placed at various locations of the actual computer memory. The algorithm is appropriate for this model as well. We have to take into account that memory is not necessarily linearly ordered as in the previous case, and consequently tests for determining where certain address belongs to should be made more strict. For example, for testing whether a node belongs to the temporary are it was sufficient to test $i \geq \mathit{temp\_first}$. In the new setting, the test should be ($i \geq \mathit{temp\_first}$ and $i \leq \mathit{temp\_last}$).

Parallel algorithm. For executing parallel algorithms, it is important to have "thread-safe" structures, i.e., multiple threads must be able to access shared data structure without race conditions. Our algorithm can be made thread-safe by using the local thread memory for keeping the stack, temporary area, and the *G-F* cells. Using a separate stack and temporary area is a straight-forward modification. For using separate *G-F* cells, Van Lohuizen's solution [16, 17] for the Tomabechi's algorithm could be used.

8 Conclusion

We have presented a computational model for graph unification in context of NL parsing. It is a novel algorithm.

The first contribution of this work is that it presents a unified, low-level algorithm that incorporates hidden structure sharing and global counter. Hidden structure sharing is not only used for non-modified nodes, but it is also recognized in situations where a node is unified with another node but it is not essentially changed. Additionally, we introduce a novel feature of sub-node hidden structure sharing, i.e., sharing of edges. According to reported information, the graph representation used here is the most compact representation until now. It also demonstrates that memory management for this task can be done directly and efficiently. This approach makes obsolete some previous arguments that assumed the use of garbage collection and frequent memory allocation system calls.

The second contribution of this work is that it shows that Wroblewski's and Tomabechi's algorithms are not significantly different, if the approach is sufficiently low-level. A difference with previous approaches is that we do not assume that union and complement of sets of edges are unit operations. This is the first paper that reveals all relevant details of graph unification for NL parsing and makes an attempt at optimizing them in a single, complete, automaton-like model.

Acknowledgments

The authors are members of the Institute for Robotics and Intelligent Systems (IRIS) and wish to acknowledge the support of the Networks of Centers of Excellence Program of the Government of Canada, the Natural Sciences and Engineering Research Council, and the participation of PRECARN Associates Inc.

References

[1] *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL-85)*, University of Chicago, Chicago, Illinois, USA, 1985.

[2] *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)*, Berkeley, California, USA, 1991.

[3] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991.

[4] R. S. Boyer and J. S. Moore. The sharing of structure in theorem proving programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 101–116. Edinburgh University Press, Edinburgh, Scotland, UK, 1972.

[5] Ulrich Callmeier. PET. A platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99–

108, March 2000. Special Issue on Efficient processing with HPSG: Methods, systems, evaluation.

[6] Ulrich Callmeier. Efficient parsing with large-scale unification grammars. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[7] Bob Carpenter. ALE user's guide. Laboratory for Computational Linguistics Report CMU-LCL-92-1, Laboratory for Computational Linguistics, Carnegie Mellon University, 1992.

[8] Bob Carpenter and Yan Qu. An abstract machine for attribute value logics. In *Proceedings of the 4th International Workshop on Parsing Technologies*, Prague, Czech Republik, 1995.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[10] Martin C. Emele. Unification with lazy non-redundant copying. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)* [2], pages 323–330. `http://citeseer.nj.nec.com/emele91unification.html`.

[11] Kurt Godden. Lazy unification. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics (ACL-90)*, pages 180–187, Pittsburgh, USA, 1990.

[12] Lauri Karttunen. D-PATR: a development environment for unification-based grammars. Technical Report CSLI-86-61, Center for the Study of Language and Information (CSLI), August 1986.

[13] Lauri Karttunen and Martin Kay. Structure sharing with binary trees. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL-85)* [1], pages 133–136A.

[14] Kevin Knight. Unification: a multidisciplinary survey. *ACM computing surveys, March 1989*, 21(1):93–124, 1989.

[15] Kiyoshi Kogure. Strategic lazy incremental copy graph unification. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*, pages 223–228, Helsinki, Finland, 1990.

[16] Marcel P. Lohuizen. Parallel unification for natural language processing. Technical Report PDS-1998-002, Delft University of Technology, 1998. Parallel and Distributed Systems Reports Series.

[17] Marcel P. Lohuizen. Memory-efficient and thread-safe quasi-destructive graph unification. In *Proceedings of the 38th Meeting of the Association for Computational Linguistics (ACL-00)*, pages 352–359, 2000.

[18] Takaki Makino, Minory Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. LiLFeS—towards a practical HPSG parser. In *Proceedigns of the COLING-ACL 98, Montreal*, pages 807–811, 1998.

[19] Takaki Makino, Minory Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. LiLFeS—towards a practical HPSG parser. In *Proceedings of the COLING-ACL 98, Montreal*, pages 807–811, 1998.

[20] Robert Malouf, John Carroll, and Ann Copestake. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1):26–46, 2000.

[21] Yusuke Miyao, Takaki Makino, Kentaro Torisawa, and Jun-Ichi Tsujii. The LiLFeS abstract machine and its evaluation with the LinGO grammar. *Natural Language Engineering*, 6(1):47–61, 2000.

[22] Fernando C. N. Pereira. A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL-85)* [1], pages 137–144.

[23] David Perelman-Hall. Directed acyclic graph unification. *Dr. Dobb's Journal*, pages 44–51, 94–99, April 1995.

[24] Stuart M. Shieber. The design of a computer language for linguistic information. In *10th International Conference on Computational Linguistics, 22nd Annual Meeting of the Association for Computational Linguistics, Proceedings of Coling84*, pages 362–366, Stanford University, California, USA, 2–6 July 1984.

[25] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford University, Stanford, CA, 1986.

[26] Hideto Tomabechi. Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)* [2], pages 315–322.

[27] David A. Wroblewski. Nondestructive graph unification. In *Proceedigns of the Sixth National Conference on Artificial Intelligence*, pages 582–587, 1987.

A Java code

```
/*
 *   $Source: /home/vkeselj/cvsroot/unifmachine/unifmachine.java,v $
 * $Revision: 1.10 $
 *     $Date: 2002/01/23 11:08:04 $
 *   Authors: (c) 2001-2002 Vlado Keselj and Nick Cercone
 */

import java.util.*;

public class unifmachine {

  static int A_len = 10000;
  static int[] A = new int[A_len]; /* the array */

  static int Alloc_last = -1;   /* last allocated cell */

  /* Attributes (features) */
  static final int att_a = -7;
  static final int att_b = -6;
  static final int att_d = -5;
  static final int att_e = -4;
  static final int att_g = -3;
  static final int att_h = -2;

  /* Atoms */
  static final int ATOM_A = -1;
```

```
static String[] att_name = { null, null, "h", "g", "e", "d", "b",
                             "a" };

static String[] atom_name = { null, "A" };

/* Global generation counter */
static int Generation = 1;

/* Root nodes bookkeeping (2 entries per root):
 *  1. entry: root address
 *  2. entry: graph size */
static int[] root = new int[10];
static int root_last = -1;

/**
   Set initial configuration.
*/
static void init() {
  Alloc_last = -1;

  /* First graph */
  root[++root_last] = Alloc_last + 1; /* address 1 */
  {
    int[] a = { 0, 0, 1, att_a,  8, att_d, 15, -1,
                0, 0, 1, att_b, 14, -1,
                ATOM_A,
                0, 0, 1, att_e, 21, -1,
                0, 0, 0 };
    System.arraycopy(a, 0, A, Alloc_last+1, a.length);
    Alloc_last += a.length;
    root[++root_last] = a.length;     /* size 1 */
  }

  /* Second graph */
  root[++root_last] = Alloc_last + 1; /* address 2 */
  {
    int[] a = { 0, 0, 1, att_a,  34, att_d, 34, att_g, 43, -1,
                0, 0, 1, att_b, 40, -1,
```

```
                0, 0, 0,
                0, 0, 1, att_h, 49, -1,
                0, 0, 0 };
    System.arraycopy(a, 0, A, Alloc_last+1, a.length);
    Alloc_last += a.length;
    root[++root_last] = a.length;     /* size 2 */
  }
}

/**
    The main function
*/
public static void main(String[] args) {

  init();
  print_latex("Initial configuration", "initialconfiguration");

  System.out.println( "%" +
                      " arg1="  + root[0] +
                      " size1=" + root[1] +
                      "  arg2=" + root[2] +
                      " size2=" + root[3] );

  int result = unify1(0, 1);

  print_latex("After first phase", "firstphasenorepackaging");

  if (result > -1)
    result = unify_copy(result);

  print_latex("Final","secondphase");

  System.out.println("% Result adr=" + root[root_last-1] + " size="
                     + root[root_last]);

  System.exit(0);

} /* end of main */
```

```
/* Non-static variables */

static int stack_top = -1;

static int temp_first = -1;   /* first temp allocated cell */
static int temp_last  = -1;   /* last temp allocated cell  */

static int repack_limit = -1;

/**
   Unify two graphs: first phase.
   @param g1 index of the first graph (2*g1 in root array)
   @param g2 index of the second graph (2*g2 in root array)
   @return -1 if unification fails,
           index of the resulting graph otherwise
*/
static int unify1(int g1, int g2) {

  /* max_size = max(2, size1 + size2 - 3) */
  int max_size = root[2*g1+1]+root[2*g2+1]-3;
  if (max_size < 2) max_size = 2;

  temp_first = 2*(root[2*g1+1]+root[2*g2+1])/3 + 1;
  if (temp_first < 2) temp_first = 2;
  temp_first += Alloc_last + 1 + max_size;
  temp_last  = temp_first - 1;
  stack_top = temp_first;
  repack_limit = A_len - 5 * max_size / 2;

  /* Check the size of the array.  Keep doubling them until the
     length is sufficient.
  */
  while (temp_last >= repack_limit) {
    int[] tmp = new int[2 * A.length];
    System.arraycopy(A, 0, tmp, 0, A.length);
    A = tmp;
```

```
  A_len = A.length;
  repack_limit = A_len - 2 * max_size;
}

A[--stack_top] = root[2*g1];
A[--stack_top] = root[2*g2];

while (stack_top < temp_first) {
  int i2 = A[stack_top++];
  int i1 = A[stack_top++];

  if (i1==i2) continue;

  /* Dereference i1 with path compression */
  if ( A[i1] == Generation ) {
    int j = A[i1+1];
    if ( A[j] == Generation ) {
      A[--stack_top] = -1;
      do {
        A[--stack_top] = i1 + 1;
        i1 = j;
        j = A[j+1];
      }
      while ( A[j] == Generation );

      while (A[stack_top] > -1)
        A[A[stack_top++]] = j;

      stack_top++;
    }

    i1 = j;
  }

  /* Dereference i2 with path compression */
  if ( A[i2] == Generation ) {
    int j = A[i2+1];
```

```
    if ( A[j] == Generation ) {
      A[--stack_top] = -1;
      do {
        A[--stack_top] = i1 + 1;
        i2 = j;
        j = A[j+1];
      }
      while ( A[j] == Generation );

      while (A[stack_top] > -1)
        A[A[stack_top++]] = j;

      stack_top++;
    }

    i2 = j;
  }

  if (i1==i2) continue;

  /* Unify */
  /* at least one node is a leaf node */
  if ( A[i1]>=0 && A[i1+2]==0 ) {
    A[i1] = Generation;
    A[i1+1] = i2;
  } else if ( A[i2]>=0 && A[i2+2]==0) {
    A[i2] = Generation;
    A[i2+1] = i1;
  }
  /* at least one node is an atom */
  else if (A[i1] < 0 || A[i2] < 0) {
    /* they have to match */
    if ( A[i1] != A[i2] ) {
      Generation++;
      return -1;            /* unification fails: atoms mismatch
                               */
    }
  }
```

```
/* otherwise, we have two complex nodes: we will merge them */
else {
  int embed1 = 0, embed2 = 0; /* can result be embedded in one
                                 of the graphs (0), otherwise,
                                 index of the attribute from
                                 which the tail can be shared */
  int src1 = i1, src2 = i2; /* shared source */

  int j1=i1+3, j2=i2+3, j3=temp_last+4; /* j1 and j2 are merged
                                           to create j3 */

  /* types should be merged here for typed feature structures */
  A[j3-1] = 1;

  /* Merge lists */
  while ( A[j1] != -1 && A[j2] != -1 ) {

    /* dereference */
    while (A[j1] >= 0) j1 = A[j1];
    while (A[j2] >= 0) j2 = A[j2];

    /* merge step */
    if ( A[j1] < A[j2] ) {
      A[j3++] = A[j1++];
      A[j3++] = A[j1++];
      embed2 = j3; src2 = j2;
    }
    else if ( A[j2] < A[j1] ) {
      A[j3++] = A[j2++];
      A[j3++] = A[j2++];
      embed1 = j3; src1 = j1;
    }
    else {
      A[j3++] = A[j1++];
      /* If values do not match? */
      if ( A[j1] != A[++j2] ) {

        /* push refs on stack */
```

```
      A[--stack_top] = A[j1];
      A[--stack_top] = A[j2];
    }
    A[j3++] = A[j1++];
    j2++;
  }
}
/* Final check for complete embedding */
if (A[j1] != -1) embed2 = -1;
else if (A[j2] != -1) embed1 = -1;

/* let's see what to do with the result */

/* Is embedding possible?  For typed feature structures,
   the resulting type should be copied. */
/* embedding into i1 */
if ( embed1==0 && (embed2!=0 || i1 <= i2 ) ) {
  A[i2++] = Generation;
  A[i2] = i1;
}
/* embedding into i2 */
else if ( embed2 == 0) {
  A[i1++] = Generation;
  A[i1] = i2;
}
else { /* no embedding */
  /* let us first decide about tail sharing */
  if ( embed1 > -1 &&
       (embed2 == -1 ||
        ( j1 <= Alloc_last && j2 > Alloc_last ) ||
        embed1 <= embed2 ) ) {
    A[ j3 = embed1 ] = src1;
    embed2 = -1;
  } else {
    A[ j3 = embed2 ] = src2;
    embed1 = -1;
  }
```

```
/* Can we leave it as it is? */
if (j3 < repack_limit ||
    ( i1 < temp_first && i2 < temp_first )) {
  A[++temp_last] = 0;
  A[i1++] = A[i2++] = Generation;
  A[i1] = A[i2] = temp_last;
  temp_last = j3;
}
/*  Else repackage */
else {
  if ( i1 < temp_first ||
       ( embed1 > -1 && i2 >= temp_first ) )
    { int tmp=i1;i1=i2;i2=tmp; }
  A[i2++] = Generation;
  A[i2++] = i1; i2++; /* let i2 be on the first edge */
  A[i1++] = 0;
  ++i1;
  j1 = temp_last + 3; /* use j1 as the source index */
  A[i1++] = A[j1++];  /* copy T */
  while (A[j1] < -1) {
    if (i1 < temp_last && A[i1] >= -1) { /* we have to find
                                            new destination */
      int ii1 = i1;
      while (A[ii1] >= 0) ii1 = A[ii1];

      if ( ii1 < temp_first ) { /* have to break the chain */
        if (i2 >= temp_first) {  /* we can switch to i2 */
          ii1 = i2; i2 = -1;
        }
        else ii1 = temp_last + 1; /* we continue after temp_last */
      }

      A[i1] = ii1; i1 = ii1;
    }
    if ( i1 == A[j3] ) { /* we hit the shared tail! */
      int ii1 = i1;
      while (A[ii1] < -1) {
        A[j3++] = A[ii1++]; A[j3++] = A[ii1++];
```

```
          }
          while ( A[ii1] > 0 ) ii1 = A[ii1];
          A[j3] = (A[ii1] == -1) ? -1 : ii1;
        }

        /* copy pair */
        A[i1++] = A[j1++];
        A[i1++] = A[j1++];
      } /* finished, just copy the last cell (-1 or ref to shared
           tail ) */
      A[i1] = A[j1];
      if ( i1 > temp_last ) temp_last = i1;

    }     /* end of else repackage */

  } /* end: no embedding */
 } /* end of complex node merging */

} /* end of unification, pop another pair from stack */

return g1;

} /* end of unify1 function */


static final int FORWARD = 1;
static final int BACKWARD = -1;
static final int END = 0;

static int unify_copy(int graph) {

  /* Copy with hidden structure sharing, using depth-first search
     Overview:
     END:   finished, final address in i
     FORWARD: go forward using address in ind and leave it in i,
     BACKWARD: go backward, last address in i

     Node statuses for nodes <= Alloc_last
```

```
   G < Generation    not visited
   G=Generation F>=0 forward
   G=Generation F<=-1 visit finished or being visited
                -F is address of original T

   set i=graph go FORWARD
*/
int i = root[2*graph];

int state = FORWARD;
int size = 0;
int new_Alloc_last = Alloc_last;

while ( state != END ) {
  /*
   * FORWARD:
   */
  if ( state == FORWARD ) {

    /* Dereference i with path compression */
    if ( A[i] == Generation && A[i+1] >= 0 ) {
      int j = A[i+1];
      if ( A[j] == Generation && A[j+1] >= 0 ) {
        A[--stack_top] = -1;
        do {
          A[--stack_top] = i + 1;
          i = j;
          j = A[j+1];
        }
        while ( A[j] == Generation && A[j+1] >= 0);

        while (A[stack_top] > -1)
          A[A[stack_top++]] = j;

        stack_top++;
      }

      i = j;
```

```
}

/* if the node is an atom, or it was visited, or being
   visited, or it is a leaf, or it does not have children and
   it is in charts area then go back */
if ( A[i] < 0 ) {
  state = BACKWARD;
  size ++;
} else if ( A[i] == Generation && A[i+1] <= -1 )
  state = BACKWARD;
else if ( i <= Alloc_last &&
          ( A[i+2] == 0 || A[i+3] == -1 ) ) {
  A[i] = Generation; A[i+1] = -1;
  size += A[i+2] == 0 ? 3 : 4;
}

/* otherwise, copy node to reserved area, with replacing F
   with negative address of F, and attributes with addresses
   of original attributes
*/
else {

  /* make a copy and the reference to it
     (size is accumulated in new_Alloc_last */
  A[++new_Alloc_last] = Generation; /* new G */
  A[i] = Generation;                /* old G */
  A[i+1] = new_Alloc_last;          /* old F */
  A[++new_Alloc_last] = - (i+1); /* new F: negative address of F */
  A[++new_Alloc_last] = A[i+2];     /* copy T */
  /* we know that it is a complex node (leaf nodes are always
     shared) */
  for (int src=i+3; ; ) {
    while ( A[src]>=0 ) src = A[src];
    if ( A[src]==-1 ) {
      A[++new_Alloc_last] = -1;
      break;
    }
    A[++new_Alloc_last] = src++; /* address of attribute */
```

```
      A[++new_Alloc_last] = A[src++];
    }
    i = A[i+1];

    /* The copy is maid, start iteration over children (if there
       are any).  Iterate backwards.
       push on stack: child ptr address
    */
    if (A[i+3]==-1) {     /* no children */
      if ( - A[i+1] <= Alloc_last ) {
        A[ - A[i+1] ] = -1; /* shared */
        new_Alloc_last -= 4;
        size += 4;
        i = - A[i+1] - 1;
      }
      state = BACKWARD;
    }
    else {
      A[--stack_top] = new_Alloc_last - 1; /* child ptr address */
      i = A[new_Alloc_last - 1];           /* child ptr */
      /* continue with state=FORWARD */
    }
  }
} /* end of state FORWARD */

/*
 * BACKWARD
 */
if ( state == BACKWARD ) {
  /* if stack is empty, it is the end */
  if (stack_top == temp_first) state = END;
  else {
    /* pop the ptr address */
    int ptr_address = A[stack_top++];

    /* can we tail share this address? */
    if ( A[ptr_address+1] >= -1 && i <= Alloc_last && i ==
         A[ptr_address] && A[ptr_address-1] <= Alloc_last ) {
```

```
          new_Alloc_last -= 2;
          size += 2;
          ptr_address -= 2;
        } else {
          A[ptr_address--] = i;
          A[ptr_address] = A[A[ptr_address]]; /* set attribute */
          ptr_address--;
        }

        if ( A[ptr_address-1] >= 0 ) { /* next child to visit */
          A[--stack_top] = ptr_address;
          i = A[ptr_address];
          state = FORWARD;
        } else {              /* node finished */
          /* can we share the whole node */
          if ( A[ptr_address+1] >= 0 &&
               -A[ptr_address-1] <= Alloc_last ) {
            new_Alloc_last -= 4;
            size += 4;
            i = - A[ ptr_address - 1 ] - 1;
            A[i+1] = -1;
          }
          else i = ptr_address - 2;
        }
      } /* BACKWARD and not END */
    } /* state BACKWARD */
  } /* while state != END */

  size += new_Alloc_last - Alloc_last;
  Alloc_last = new_Alloc_last;
  temp_first = temp_last = -1;
  root[++root_last] = i;
  root[++root_last] = size;
  ++Generation;
  return root_last - 1;
}

static void print_a() {
```

```
  for (int i=0; i < A.length; ++i) {
    if (i>Alloc_last && i > temp_last) break;
    System.out.print("a["+spacePadded(i,3)+"]="+spacePadded(A[i],-4));
    if (i % 10 == 9) System.out.println();
  }
  System.out.println();
}

/**
   Auxiliary function -- space padding on left or right to an integer.
   @param i an integer to be represented as astring
   @param pad the min length, if negative pad on right, otherwise
   pad on left
   @return string representation
*/
static String spacePadded(int i, int pad) {
  String r = "" + i;
  int apad = (pad < 0 ? -pad : pad);
  while (r.length() < apad)
    r = (pad<0 ? r+" " : " "+r);
  return r;
}

/* kind_of_cell: 0 - not used, 1 - GFT, 2 - attribute */
static void mark_gft(int i, int[] mark) {
  if (mark[i] != 0) return;
  mark[i] = 1;
  if (A[i] == Generation && A[i+1] >= 0) mark_gft(A[i+1], mark);
  else if (A[i] >=0 && A[i+2]!=0) mark_att(i+3, mark);
}
static void mark_att(int i, int[] mark) {
  if (mark[i] != 0) return;
  mark[i] = 2;
  if ( A[i] >= 0 ) mark_att( A[i], mark );
  else if (A[i] < -1) {
    mark_gft(A[i+1], mark);
    mark_att(i+2, mark);
  }
```

```
}

/**
   Print contents of the array in LaTeX form.
 */
static void print_latex(String comment, String command) {

  int[] kind_of_cell = new int[ (temp_last > Alloc_last ?
                                temp_last : Alloc_last) + 1 ];

  /*
   * Print out up to Alloc_last and mark first GFT's
   */

  System.out.println("% " + comment + "\n");

  System.out.println("\\newcommand{\\" + command + "}{" +
                     "\\setcounter{cell}{0}\\raggedright");

  for (int i=0; i <= Alloc_last; ) {
    mark_gft(i, kind_of_cell);
    i = print_latex_gft( i );
  }

  System.out.println("\\\\$\\mbox{\\it Alloc\\_last}="+Alloc_last+"$");

  if (temp_first > Alloc_last && temp_last > Alloc_last) {

    System.out.println("\\quad\\setcounter{cell}{"+temp_first+
                     "}$\\mbox{\\it temp\\_first}="+temp_first+"$\\\\");

    /* Print out the temporary area */
    for (int i = temp_first; i <= temp_last; ) {
      switch ( kind_of_cell[i] ) {
      case 1: i = print_latex_gft( i ); break;
      case 2: i = print_latex_att( i ); break;
      case 0:
        System.out.println("\\cells{"+A[i]+"}{}% " + i);
```

```
        ++i;
      }
    }

  }

  System.out.println("}\n");

} /* end of function print_latex */

/** Print continous part of the node and return next i */
static int print_latex_gft( int i ) {

  if ( A[i] < 0 )               /* atom */
    System.out.println("\\celle{"+ atom_name[-A[i++]] + "} % " +
                       (i-1));
  else if (A[i] == Generation && i > Alloc_last)   /* forward */
    System.out.println("\\cell{" + A[i++] + "}{G}\\cell{" + A[i++] +
                       "}{F} % " + (i-2) );
  else {                        /* gft ... */
    int bg = i;
    System.out.print("\\cellgft{"+A[i++]+"}{"+A[i++]+"}{"+A[i++]+"}");
    if ( A[i-1] == 0 )          /* leaf */
      System.out.println(" % " + bg);
    else {
      i = print_latex_att( i );
      System.out.println(" % " + bg);
    }
  }

  return i;
}

/** Print continous edge sequence, return next i */
static int print_latex_att( int i ) {
  while ( A[i] < -1 ) {
    System.out.print("\\cellp{" + att_name[ -A[i++] ] + "}{" +
                     A[i++] + "}");
```

```
    }
    System.out.print("\\celle{" + A[i++] + "}");
    return i;
  }

} /* end of class */
```

B C code

```
/*
 *   $Source: /home/vkeselj/cvsroot/unifmachine/unifmachine.c,v $
 * $Revision: 1.3 $
 *     $Date: 2002/01/23 11:08:04 $
 *   Authors: (c) 2001-2002 Vlado Keselj and Nick Cercone
 */

#include <stdio.h>

void mark_att(int i, int *mark);
void print_latex(char* comment, char* command);

#define A_len 10000
int A[A_len];           /* the array */

int Alloc_last = -1;    /* last allocated cell */

/* Attributes (features) */
#define att_a -7
#define att_b -6
#define att_d -5
#define att_e -4
#define att_g -3
#define att_h -2

/* Atoms */
#define ATOM_A -1
```

```
char* att_name[] = { NULL, NULL, "h", "g", "e", "d", "b",
                     "a" };

char* atom_name[] = { NULL, "A" };

/* Global generation counter */
int Generation = 1;

/* Root nodes bookkeeping (2 entries per root):
 *  1. entry: root address
 *  2. entry: graph size */
int root[10];
int root_last = -1;

/**
   Set initial configuration.
*/
void init() {
  Alloc_last = -1;

  /* First graph */
  root[++root_last] = Alloc_last + 1; /* address 1 */
  {
    int a[] = { 0, 0, 1, att_a,  8, att_d, 15, -1,
                0, 0, 1, att_b, 14, -1,
                ATOM_A,
                0, 0, 1, att_e, 21, -1,
                0, 0, 0 }; /* 24 elements */
    int i;
    for (i=0; i < 24; ++i)
      A[Alloc_last+1+i] = a[i];
    Alloc_last += 24;
    root[++root_last] = 24;     /* size 1 */
  }

  /* Second graph */
  root[++root_last] = Alloc_last + 1; /* address 2 */
  {
```

```
    int a[] = { 0, 0, 1, att_a,  34, att_d, 34, att_g, 43, -1,
                0, 0, 1, att_b, 40, -1,
                0, 0, 0,
                0, 0, 1, att_h, 49, -1,
                0, 0, 0 };      /* 28 */
    int i;
    for (i=0; i < 28; ++i)
      A[Alloc_last+1+i] = a[i];
    Alloc_last += 28;
    root[++root_last] = 28;     /* size 2 */
  }
}

/**
    The main function
*/
int main(int argc, char *argv[]) {

  int result;

  init();
  print_latex("Initial configuration", "initialconfiguration");

  printf("%% arg1=%d size1=%d  arg2=%d size2=%d\n", root[0], root[1],
         root[2], root[3]);

  result = unify1(0, 1);

  print_latex("After first phase", "firstphasenorepackaging");

  if (result > -1)
    result = unify_copy(result);

  print_latex("Final","secondphase");

  printf("%% Result adr=%d size=%d\n", root[root_last-1],
         root[root_last]);
```

```
  return 0;

} /* end of main */


/* Non-static variables */

int stack_top = -1;

int temp_first = -1;    /* first temp allocated cell */
int temp_last  = -1;    /* last temp allocated cell  */

int repack_limit = -1;

/**
   Unify two graphs: first phase.
   @param g1 index of the first graph (2*g1 in root array)
   @param g2 index of the second graph (2*g2 in root array)
   @return -1 if unification fails,
   index of the resulting graph otherwise
*/
int unify1(int g1, int g2) {

    /* max_size = max(2, size1 + size2 - 3) */
    int max_size = root[2*g1+1]+root[2*g2+1]-3;
    if (max_size < 2) max_size = 2;

    temp_first = 2*(root[2*g1+1]+root[2*g2+1])/3 + 1;
    if (temp_first < 2) temp_first = 2;
    temp_first += Alloc_last + 1 + max_size;
    temp_last  = temp_first - 1;
    stack_top = temp_first;
    repack_limit = A_len - 5 * max_size / 2;

    /* Check the size of the array.  */
    if (temp_last >= repack_limit) {
      fprintf(stderr, "Insufficient memory");
      exit(1);
```

```
}

A[--stack_top] = root[2*g1];
A[--stack_top] = root[2*g2];

while (stack_top < temp_first) {
  int i2 = A[stack_top++];
  int i1 = A[stack_top++];

  if (i1==i2) continue;

  /* Dereference i1 with path compression */
  if ( A[i1] == Generation ) {
    int j = A[i1+1];
    if ( A[j] == Generation ) {
      A[--stack_top] = -1;
      do {
        A[--stack_top] = i1 + 1;
        i1 = j;
        j = A[j+1];
      }
      while ( A[j] == Generation );

      while (A[stack_top] > -1)
        A[A[stack_top++]] = j;

      stack_top++;
    }

    i1 = j;
  }

  /* Dereference i2 with path compression */
  if ( A[i2] == Generation ) {
    int j = A[i2+1];
    if ( A[j] == Generation ) {
      A[--stack_top] = -1;
      do {
```

```
      A[--stack_top] = i1 + 1;
      i2 = j;
      j = A[j+1];
    }
    while ( A[j] == Generation );

    while (A[stack_top] > -1)
      A[A[stack_top++]] = j;

    stack_top++;
  }

  i2 = j;
}

if (i1==i2) continue;

/* Unify */
/* at least one node is a leaf node */
if ( A[i1]>=0 && A[i1+2]==0 ) {
  A[i1] = Generation;
  A[i1+1] = i2;
} else if ( A[i2]>=0 && A[i2+2]==0) {
  A[i2] = Generation;
  A[i2+1] = i1;
}
/* at least one node is an atom */
else if (A[i1] < 0 || A[i2] < 0) {
  /* they have to match */
  if ( A[i1] != A[i2] ) {
    Generation++;
    return -1;            /* unification fails: atoms mismatch
                           */
  }
}
/* otherwise, we have two complex nodes: we will merge them */
else {
  int embed1 = 0, embed2 = 0; /* can result be embedded in one
```

```
                                   of the graphs (0), otherwise,
                                   index of the attribute from
                                   which the tail can be shared */
int src1 = i1, src2 = i2; /* shared source */

int j1=i1+3, j2=i2+3, j3=temp_last+4; /* j1 and j2 are merged
                                         to create j3 */

/* types should be merged here for typed feature structures */
A[j3-1] = 1;

/* Merge lists */
while ( A[j1] != -1 && A[j2] != -1 ) {

  /* dereference */
  while (A[j1] >= 0) j1 = A[j1];
  while (A[j2] >= 0) j2 = A[j2];

  /* merge step */
  if ( A[j1] < A[j2] ) {
    A[j3++] = A[j1++];
    A[j3++] = A[j1++];
    embed2 = j3; src2 = j2;
  }
  else if ( A[j2] < A[j1] ) {
    A[j3++] = A[j2++];
    A[j3++] = A[j2++];
    embed1 = j3; src1 = j1;
  }
  else {
    A[j3++] = A[j1++];
    /* If values do not match? */
    if ( A[j1] != A[++j2] ) {

      /* push refs on stack */
      A[--stack_top] = A[j1];
      A[--stack_top] = A[j2];
    }
```

```
    A[j3++] = A[j1++];
    j2++;
  }
}
/* Final check for complete embedding */
if (A[j1] != -1) embed2 = -1;
else if (A[j2] != -1) embed1 = -1;

/* let's see what to do with the result */

/* Is embedding possible?  For typed feature structures,
   the resulting type should be copied. */
/* embedding into i1 */
if ( embed1==0 && (embed2!=0 || i1 <= i2 ) ) {
  A[i2++] = Generation;
  A[i2] = i1;
}
/* embedding into i2 */
else if ( embed2 == 0) {
  A[i1++] = Generation;
  A[i1] = i2;
}
else { /* no embedding */
  /* let us first decide about tail sharing */
  if ( embed1 > -1 &&
       (embed2 == -1 ||
        ( j1 <= Alloc_last && j2 > Alloc_last ) ||
        embed1 <= embed2 ) ) {
    A[ j3 = embed1 ] = src1;
    embed2 = -1;
  } else {
    A[ j3 = embed2 ] = src2;
    embed1 = -1;
  }

  /* Can we leave it as it is? */
  if (j3 < repack_limit ||
      ( i1 < temp_first && i2 < temp_first )) {
```

```
    A[++temp_last] = 0;
    A[i1++] = A[i2++] = Generation;
    A[i1] = A[i2] = temp_last;
    temp_last = j3;
  }
  /*  Else repackage */
  else {
    if ( i1 < temp_first ||
         ( embed1 > -1 && i2 >= temp_first ) )
      { int tmp=i1;i1=i2;i2=tmp; }
    A[i2++] = Generation;
    A[i2++] = i1; i2++; /* let i2 be on the first edge */
    A[i1++] = 0;
    ++i1;
    j1 = temp_last + 3; /* use j1 as the source index */
    A[i1++] = A[j1++];  /* copy T */
    while (A[j1] < -1) {
      if (i1 < temp_last && A[i1] >= -1) { /* we have to find
                                               new destination */
        int ii1 = i1;
        while (A[ii1] >= 0) ii1 = A[ii1];

        if ( ii1 < temp_first ) { /* have to break the chain */
          if (i2 >= temp_first) {  /* we can switch to i2 */
            ii1 = i2; i2 = -1;
          }
          else ii1 = temp_last + 1; /* we continue after temp_last */
        }

        A[i1] = ii1; i1 = ii1;
      }
      if ( i1 == A[j3] ) { /* we hit the shared tail! */
        int ii1 = i1;
        while (A[ii1] < -1) {
          A[j3++] = A[ii1++]; A[j3++] = A[ii1++];
        }
        while ( A[ii1] > 0 ) ii1 = A[ii1];
        A[j3] = (A[ii1] == -1) ? -1 : ii1;
```

```
          }

          /* copy pair */
          A[i1++] = A[j1++];
          A[i1++] = A[j1++];
        } /* finished, just copy the last cell (-1 or ref to shared
             tail ) */
        A[i1] = A[j1];
        if ( i1 > temp_last ) temp_last = i1;

      }     /* end of else repackage */

    } /* end: no embedding */
  } /* end of complex node merging */

} /* end of unification, pop another pair from stack */

return g1;

} /* end of unify1 function */


#define FORWARD  1
#define BACKWARD -1
#define END      0

int unify_copy(int graph) {

    /* Copy with hidden structure sharing, using depth-first search
       Overview:
       END:   finished, final address in i
       FORWARD: go forward using address in ind and leave it in i,
       BACKWARD: go backward, last address in i

       Node statuses for nodes <= Alloc_last
       G < Generation    not visited
       G=Generation F>=0 forward
       G=Generation F<=-1 visit finished or being visited
```

```
                 -F is address of original T

   set i=graph go FORWARD
*/
int i = root[2*graph];

int state = FORWARD;
int size = 0;
int new_Alloc_last = Alloc_last;

while ( state != END ) {
  /*
   * FORWARD:
   */
  if ( state == FORWARD ) {

    /* Dereference i with path compression */
    if ( A[i] == Generation && A[i+1] >= 0 ) {
      int j = A[i+1];
      if ( A[j] == Generation && A[j+1] >= 0 ) {
        A[--stack_top] = -1;
        do {
          A[--stack_top] = i + 1;
          i = j;
          j = A[j+1];
        }
        while ( A[j] == Generation && A[j+1] >= 0);

        while (A[stack_top] > -1)
          A[A[stack_top++]] = j;

        stack_top++;
      }

      i = j;
    }

    /* if the node is an atom, or it was visited, or being
```

```
     visited, or it is a leaf, or it does not have children and
     it is in charts area then go back */
  if ( A[i] < 0 ) {
    state = BACKWARD;
    size ++;
  } else if ( A[i] == Generation && A[i+1] <= -1 )
    state = BACKWARD;
  else if ( i <= Alloc_last &&
            ( A[i+2] == 0 || A[i+3] == -1 ) ) {
    A[i] = Generation; A[i+1] = -1;
    size += A[i+2] == 0 ? 3 : 4;
  }

  /* otherwise, copy node to reserved area, with replacing F
     with negative address of F, and attributes with addresses
     of original attributes
  */
  else {
    int src;

    /* make a copy and the reference to it
       (size is accumulated in new_Alloc_last */
    A[++new_Alloc_last] = Generation; /* new G */
    A[i] = Generation;                /* old G */
    A[i+1] = new_Alloc_last;          /* old F */
    A[++new_Alloc_last] = - (i+1); /* new F: negative address of F */
    A[++new_Alloc_last] = A[i+2];     /* copy T */
    /* we know that it is a complex node (leaf nodes are always
       shared) */
    for (src=i+3; ; ) {
      while ( A[src]>=0 ) src = A[src];
      if ( A[src]==-1 ) {
        A[++new_Alloc_last] = -1;
        break;
      }
      A[++new_Alloc_last] = src++; /* address of attribute */
      A[++new_Alloc_last] = A[src++];
    }
```

```
    i = A[i+1];

    /* The copy is maid, start iteration over children (if there
       are any).  Iterate backwards.
       push on stack: child ptr address
    */
    if (A[i+3]==-1) {     /* no children */
      if ( - A[i+1] <= Alloc_last ) {
        A[ - A[i+1] ] = -1; /* shared */
        new_Alloc_last -= 4;
        size += 4;
        i = - A[i+1] - 1;
      }
      state = BACKWARD;
    }
    else {
      A[--stack_top] = new_Alloc_last - 1; /* child ptr address */
      i = A[new_Alloc_last - 1];           /* child ptr */
      /* continue with state=FORWARD */
    }
  }
} /* end of state FORWARD */

/*
 * BACKWARD
 */
if ( state == BACKWARD ) {
  /* if stack is empty, it is the end */
  if (stack_top == temp_first) state = END;
  else {
    /* pop the ptr address */
    int ptr_address = A[stack_top++];

    /* can we tail share this address? */
    if ( A[ptr_address+1] >= -1 && i <= Alloc_last && i ==
         A[ptr_address] && A[ptr_address-1] <= Alloc_last ) {
      new_Alloc_last -= 2;
      size += 2;
```

```
          ptr_address -= 2;
        } else {
          A[ptr_address--] = i;
          A[ptr_address] = A[A[ptr_address]]; /* set attribute */
          ptr_address--;
        }

        if ( A[ptr_address-1] >= 0 ) { /* next child to visit */
          A[--stack_top] = ptr_address;
          i = A[ptr_address];
          state = FORWARD;
        } else {              /* node finished */
          /* can we share the whole node */
          if ( A[ptr_address+1] >= 0 &&
               -A[ptr_address-1] <= Alloc_last ) {
            new_Alloc_last -= 4;
            size += 4;
            i = - A[ ptr_address - 1 ] - 1;
            A[i+1] = -1;
          }
          else i = ptr_address - 2;
        }
      } /* BACKWARD and not END */
    } /* state BACKWARD */
  } /* while state != END */

  size += new_Alloc_last - Alloc_last;
  Alloc_last = new_Alloc_last;
  temp_first = temp_last = -1;
  root[++root_last] = i;
  root[++root_last] = size;
  ++Generation;
  return root_last - 1;
}

void print_a() {
  int i;
  for (i=0; i < A_len; ++i) {
```

```
    if (i>Alloc_last && i > temp_last) break;
    printf("a[%3d]=%-4d ", i, A[i]);
    if (i % 10 == 9) putchar('\n');
  }
  putchar('\n');
}


/* kind_of_cell: 0 - not used, 1 - GFT, 2 - attribute */
void mark_gft(int i, int *mark) {
  if (mark[i] != 0) return;
  mark[i] = 1;
  if (A[i] == Generation && A[i+1] >= 0) mark_gft(A[i+1], mark);
  else if (A[i] >=0 && A[i+2]!=0) mark_att(i+3, mark);
}

void mark_att(int i, int *mark) {
  if (mark[i] != 0) return;
  mark[i] = 2;
  if ( A[i] >= 0 ) mark_att( A[i], mark );
  else if (A[i] < -1) {
      mark_gft(A[i+1], mark);
      mark_att(i+2, mark);
  }
}

/**
   Print contents of the array in LaTeX form.
*/
int kind_of_cell[A_len];
void print_latex(char* comment, char* command) {
  int i;
  for (i=0; i<A_len; ++i)
    kind_of_cell[i] = 0;

  /*
   * Print out up to Alloc_last and mark first GFT's
   */
```

```
  printf("%% %s\n\n", comment);
  printf("\\newcommand{\\%s}{\\setcounter{cell}{0}\\raggedright\n", command);

  for (i=0; i <= Alloc_last; ) {
    mark_gft(i, kind_of_cell);
    i = print_latex_gft( i );
  }

  printf("\\\\$\\mbox{\\it Alloc\\_last}=%d$\n", Alloc_last);

  if (temp_first > Alloc_last && temp_last > Alloc_last) {

    printf("\\quad\\setcounter{cell}{%d}$\\mbox{\\it temp\\_first}=%d$\\\\\n",
           temp_first, temp_first);

    /* Print out the temporary area */
    for (i = temp_first; i <= temp_last; ) {
      switch ( kind_of_cell[i] ) {
      case 1: i = print_latex_gft( i ); break;
      case 2: i = print_latex_att( i ); break;
      case 0:
        printf("\\cells{%d}{}% %d\n", A[i], i);
        ++i;
      }
    }

  }

  printf("%s", "}\n\n");

} /* end of function print_latex */

/** Print continous part of the node and return next i */
int print_latex_gft( int i ) {

  if ( A[i] < 0 ) {              /* atom */
    printf("\\celle{%s} %% %d\n", atom_name[-A[i]], i);
    ++i;
```

```
  }
  else if (A[i] == Generation && i > Alloc_last) {  /* forward */
    printf("\\cell{%d}{G}", A[i++]);
    printf("\\cell{%d}{F} %% %d\n", A[i++], i-2);
  }
  else {                        /* gft ... */
    int bg = i;
    printf("\\cellgft{%d}", A[i++]);
    printf("{%d}", A[i++]);
    printf("{%d}", A[i++]);
    if ( A[i-1] == 0 )  /* leaf */
      printf(" %% %d\n", bg);
    else {
      i = print_latex_att( i );
      printf(" %% %d\n", bg);
    }
  }

  return i;
}

/** Print continous edge sequence, return next i */
int print_latex_att( int i ) {
  while ( A[i] < -1 ) {
    printf("\\cellp{%s}", att_name[ -A[i++] ]);
    printf("{%d}", A[i++]);
  }
  printf("\\celle{%d}", A[i++] );
  return i;
}
```