Unification of Disjunctive Feature Descriptions

Andreas Eisele, Jochen Dörre
Institut für Maschinelle Sprachverarbeitung
Universität Stuttgart
Keplerstr. 17, 7000 Stuttgart 1, West Germany
Netmail: ims@rusvx2.rus.uni-stuttgart.dbp.de

Abstract

The paper describes a new implementation of feature structures containing disjunctive values, which can be characterized by the following main points: Local representation of embedded disjunctions, avoidance of expansion to disjunctive normal form and of repeated test-unifications for checking consistence. The method is based on a modification of Kasper and Rounds' calculus of feature descriptions and its correctness therefore is easy to see. It can handle cyclic structures and has been incorporated successfully into an environment for grammar development.

1 Motivation

In current research in computational linguistics but also in extralinguistic fields unification has turned out to be a central operation in the modelling of data types or knowledge in general.

Among linguistic formalisms and theories which are based on the unification paradigm are such different theories as FUG [Kay 79,Kay 85], LFG [Kaplan/Bresnan 82], GSPG [Gazdar et al. 85], CUG [Uszkoreit 86]. However, research in unification is also relevant for fields like logic programming, theorem proving, knowledge representation (see [Smolka/Aït-Kaci 87] for multiple inheritance hierarchies using unification), programming language design [Aït-Kaci/Nasr 86] and others.

The version of unification our work is based on is *graph unification*, which is an extension of *term unification*. In graph unification the number of arguments is free and arguments are selected by attribute labels rather than by position. The algorithm described here may easily be modified to apply to term unification.

The structures we are dealing with are rooted directed graphs where arcs starting in one node must carry distinct labels. Terminal nodes may also be labelled. These structures are referred to by various names in the literature: feature structures, functional structures, functional descriptions, types, categories. We will call them feature structures[1] throughout this paper.

In applications, other than toy applications, the efficient processing of *indefinite information* which is represented by *disjunctive specifications* becomes a relevant factor. A strategy of multiplying-out disjunction by exploiting (nearly) any combination of disjuncts through backtracking, as it is done, e.g., in the case of a simple DCG parser, quickly runs into efficiency problems. On the other hand the descriptional power of disjunction often helps to state highly ambiguous linguistic knowledge clearly and concisely (see Fig. 1 for a disjunctive description of morphological features for the six readings of the german noun 'Koffer').

Koffer:

$$\begin{bmatrix} \text{morph:} & \left\{ \begin{bmatrix} \text{agr:} & \begin{bmatrix} \text{num:} & \text{sg} \\ \text{pers:} & 3 \end{bmatrix} \\ \text{gend:} & \text{masc} \\ \text{case:} & \{\text{nom dat acc}\} \end{bmatrix} \right. \\ & \left. \begin{bmatrix} \text{agr:} & \begin{bmatrix} \text{num:} & \text{pl} \\ \text{pers:} & 3 \end{bmatrix} \\ \text{gend:} & \text{masc} \\ \text{case:} & \{\text{nom gen acc}\} \end{bmatrix} \right\} \\ \text{sem:} & \begin{bmatrix} \text{rel:} & \text{Koffer}' \\ \text{arg:} & [\,] \end{bmatrix} \\ \ldots & \end{bmatrix}$$

Figure 1: Using disjunction in the description of linguistic structures

Kasper and Rounds [86] motivated the distinction between feature structures and formulae of a logical calculus that are used to describe feature structures. *Disjunction* can be used within such a formula to describe *sets* of feature structures. With this separation the underlying mathematical framework which is used to define the semantics of the descriptions can be kept simple.

[1] We do not, as is frequently done, restrict ourselves to acyclic structures.

2 Disjunctive Feature Descriptions

We use a slightly modified version of the formula language FML of Kasper and Rounds [86] to describe our feature structures. Fig. 2 gives the syntax of FML′, where A is the set of atoms and L the set of labels.

FML′ contains:

NIL	
TOP	
a	where $a \in A$
$l : \Phi$	where $l \in L, \Phi \in$ FML′
$\Phi \wedge \Psi$	where $\Phi, \Psi \in$ FML′
$\Phi \vee \Psi$	where $\Phi, \Psi \in$ FML′
$\langle p \rangle$	where $p \in L^*$

Figure 2: Syntax of FML′

In contrast to Kasper and Rounds [86] we do not use the syntactic construct of path equivalence classes. Instead, path equivalences are expressed using non-local path expressions (called pointers in the sequel). This choice is motivated by the fact that we use these pointers for an efficient representation below, and we want to keep FML′ as simple as possible.

The intuitive semantics of FML′ is as follows (see [Kasper/Rounds 86] for formal definitions):

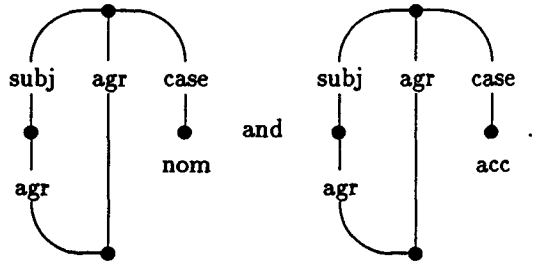
1. NIL is satisfied by any feature structure.
2. TOP is never satisfied.
3. a is satisfied by the feature structure consisting only of a single node labelled a.
4. $l : \Phi$ requires a (sub-)structure under arc l to satisfy Φ.
5. $\Phi \wedge \Psi$ is satisfied by a feature structure that satisfies Φ and satisfies Ψ.
6. $\Phi \vee \Psi$ is satisfied by a feature structure that satisfies Φ or satisfies Ψ.
7. $\langle p \rangle$ requires a path equivalence (two paths leading to the same node) between the path $\langle p \rangle$ and the actual path relative to the top-level structure.[2]

The denotation of a formula Φ is usually defined as the set of minimal elements of SAT(Φ) with respect to subsumption[3], where SAT(Φ) is the set of feature structures which satisfy Φ.

Example: The formula

Φ = subj:agr:⟨agr⟩ ∧ case:(nom ∨ acc)

denotes the two graphs



3 The Problem

The unification problem for disjunctive feature descriptions can be stated as follows:

> Given two formulae that describe feature structures, find the set of feature structures that satisfy both formulae, if it is nonempty, else announce 'fail'.

The simplest way to deal with disjunction is to rewrite any description into disjunctive normal form (DNF). This transformation requires time and space exponential with the number of disjuncts in the initial formula in the worst case. Although the problem of unifying disjunctive descriptions is known to be NP-complete (see [Kasper 87a]), methods which avoid this transformation may perform well in most practical cases. The key idea is to keep disjunction local and consider combinations of disjuncts only when they refer to the very same substructure. This strategy, however, is complicated by the fact that feature structures may be graphs with path equivalences and not only trees. Fig. 3 shows an example where unifying a disjunction with a structure containing reentrancy causes parts of the disjunction to be linked to other parts of the structure. The disjunction is *exported* via this reentrancy. Hence, the value of attribute d cannot be represented uniquely. It may be + or -, depending on which disjunct in attribute a is chosen. To represent this information without extra formal devices we have to lift the disjunction one level up.[4]

[2] This construct is context-sensitive in the sense that the denotation of $\langle p \rangle$ may only be computed with respect to the whole structure that the formula describes.

[3] The subsumption relation $\sqsubseteq$ is a partial ordering on feature structures inducing a semi-lattice. It may be defined as: FS1 $\sqsubseteq$ FS2 iff the set of formulae satisfied by FS2 includes the set of formulae satisfied by FS1.

[4] In this special case we still could keep the disjunction in the attribute a by inverting the pointer. A pointer (a b) underneath label d would allow us to specify the value of d dependent on the disjunction under a.

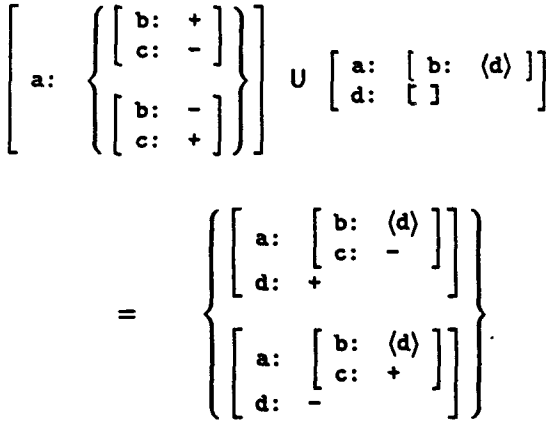


Figure 3: Lifting of disjunction due to reentrancy

4 From Description to Efficient Representation

It is interesting to investigate whether FML′ is suitable as an encoding of feature structures, i.e. if it can be used for computational purposes.

However, this is clearly not the case for the unrestricted set of formulae of FML′, since a given feature structure can be represented by infinitely many different formulae of arbitrary complexity and — even worse — because it is also not possible to ascertain whether a given formula represents any feature structure at all without extensive computation.

On the other hand, the formulae of FML′ have some properties that are quite attractive for representing feature structures, such as embedded and general disjunction and the possibility to make use of the law of distributivity for disjunctions.

Therefore we have developed an efficiency-oriented normal form ENF, which is suitable as an efficient representation for sets of feature structures.

The formulae are built according to a restricted syntax (Fig. 4, Part A) and have to satisfy condition C_{ENF} (Part B). The syntax restricts the use of conjunction and TOP in order to disallow contradictory information in a formula other than TOP. However, even in a formula of the syntax of Part A inconsistence can be introduced by a pointer to a location that is 'blocked' by an atomic value on a higher level. For example in the formula a:⟨b c⟩ ∧ b:d the path ⟨b c⟩ is blocked since it would require the value of attribute b to be complex in conflict to the atomic value d, thus rendering the

A) Restricted syntax of ENF:

NIL	
TOP	
a	where $a \in A$
$l_1 : \Phi_1 \wedge \cdots \wedge l_n : \Phi_n$	where $\Phi_i \in$ ENF$\backslash\{$TOP$\}$, $l_i \in L$, $l_i \neq l_j$ for $i \neq j$
$\Phi \vee \Psi$	where $\Phi, \Psi \in$ ENF$\backslash\{$TOP$\}$
$\langle p \rangle$	where $p \in L^*$.

B) Additional condition C_{ENF}:

If an *instance* ϕ of a formula Φ contains a pointer $\langle p \rangle$, then the path p must be *realized* in ϕ.

Figure 4: A normal form to describe feature structures efficiently

formula non-satisfiable. With the additional condition C_{ENF} such inconsistencies are excluded. Its explanation in the next section is somewhat technical and is not prerequisite for the overall understanding of our method.

Condition C_{ENF}

First we have to introduce some terminology.

Instance: When every disjunction in a formula is replaced by one of its disjuncts, the result is called an *instance* of that formula.

Realized: A recursive definition of what we call a *realized path* in an instance ϕ is given in Fig. 5. The intuitive idea behind this notion is to restrict

ε	is realized in ϕ, if $\phi \neq$ TOP
$l \in L$	is realized in $l_1 : \phi_1 \wedge \cdots \wedge l_n : \phi_n$ (even if $l \notin \{l_1 \ldots l_n\}$)
$l \cdot p$	is realized in $\cdots \wedge l : \phi \wedge \cdots$, if p is realized in ϕ
p	is realized in $\langle p' \rangle$, if $p'p$ is realized in the top-level formula

Figure 5: Definition of realized paths

pointers in such a way that the path to their destination may not be blocked by the introduction of an atomic value on a prefix of this path. Note that by virtue of the second line of the definition, the last label of the path does not have to actually occur in the formula, if there are other labels.

Example: In a:⟨b c⟩ only the path ε and each path of length 1 is realized. Any longer path may be blocked by the introduction of an atomic value at level 1. Thus, the formula violates C_{ENF}.

a:⟨b d⟩ ∧ b:⟨c⟩ ∧ c:(d:x ∨ b:y), on the other hand, is a well-formed ENF formula, since it contains only pointers with realized destinations in every disjunct.

The easiest way to satisfy the condition is to introduce for each pointer the value NIL at its destination when building up a formula. With this strategy we actually never have to check this condition, since it is maintained by the unification algorithm described below.

Properties of ENF

The most important properties of formulae in ENF are:

- For each formula of FML′ an equivalent formula in ENF can be found.
- Each instance of a formula in ENF (besides TOP) denotes exactly one feature structure.
- This feature structure can be computed in linear time.

The first property can be established by virtue of the unification algorithm given in the next section, which can be used to construct an equivalent ENF-formula for an arbitrary formula in FML′.

The next point says: It doesn't matter which disjunct in one disjunction you choose — you cannot get a contradiction. Disjunctions in ENF are *mutually independent*. This also implies that TOP is the only formula in ENF that is not satisfiable. To see why this property holds, first consider formulae without pointers. Contradictory information (besides TOP) can only be stated using conjunction. But since we only allow conjunctions of different attributes, inconsistent information cannot be stated in formulae without pointers.

Pointers could introduce two sorts of inconsistencies: Since a pointer links two paths, one might assume that inconsistent information could be specified for them. But since conjunction with a pointer is not allowed, only the destination path can carry additional information, thus excluding this kind of inconsistency. On the other hand, pointers imply the existence of the paths they refer to. The condition C_{ENF} ensures that no information in the formula contradicts the introduction of these implied paths. We can conclude that even formulae containing pointers are consistent.

The condition C_{ENF} additionally requires that no extension of a formula, gained by unification with another formula, may contain such contradicting information. A unification algorithm thus can introduce an atomic value into a formula without having to check if it would block the destination path of some pointer.

5 The Unification Procedure

Figure 6 shows an algorithm that takes as input two terms representing formulae in ENF and computes an ENF-representation of their unification. The representation of the formulae is given by a 1-to-1-mapping between formulae and datastructures, so that we can abstract from the datastructures and write formulae instead. In this sense, the logical connectives $\wedge, \vee, :$ are used as term-constructors that build more complex datastructures from simpler ones. In addition, we use the operator $\cdot$ to express concatenation of labels or label sequences and write $\langle p \rangle$ to express the pointer to the location specified by the label sequence p. $p : \Phi$ is an abbreviation for a formula where the subformula Φ is embedded on path p.

The auxiliary function `unify_aux` performs the essential work of the unification. It traverses both formulae in parallel and builds all encountered subformulae into the output formula. The following cases have to be considered:

- If one of the input formulae specifies a subformula at a location where the other input provides no information *or* if both inputs contain the same subformula at a certain location, this subformula is built into the output without modification.
- The next statement handles the case where one input contains a pointer whereas the other contains a different subformula. Since we regard the destination of the pointer as the representative of the equivalence class of paths, the subformula has to be moved to that place. This case requires additional discussion, so we have moved it to the procedure `move_formula`.
- In case of two conjunctions the formulae have to be traversed recursively and all resulting attribute - value pairs have to be built into the output structure. For clarity, this part of the algorithm has been moved to the procedure `unify_complex`.
- The case where one of the input formulae is a disjunction is handled in the procedure `unify_disj` that is described in Section 5.2.
- If none of the previous cases matches (e.g. if the inputs are different atoms or an atom and a complex formula), a failure of the unification has to be announced which is done in the last

```
unify(X,Y) ↦ formula
  repeat
      (X,Y) := unify_aux(X,Y,ε)
  until Y = NIL or Y = TOP
  return(X)


unify_aux(A0,A1,Pa) ↦ (formula,formula)
  if A0 = A1 then
      return (A1,NIL)
  else if Ai = NIL then
      return (A1-i,NIL)
  else if Ai is the pointer <Pto> then
      return move_formula(A1-i,Pa,Pto)
  else if both Ai are conjunctions then
      return unify_complex(A0,A1,Pa)
  else if Ai is the disjunction (B ∨ C)
      then
         return unify_disj(A1-i,B,C,Pa)
  else return (TOP,TOP)


unify_complex(A0,A1,Pa)
                  ↦ (formula,formula)
  L := ⋀ l:v, where l:v occurs in one Ai
            and l does not occur in A1-i
  G := NIL
  for all l that appear in both Ai do
    let V0,V1 be the values of l in A0,A1
    (V,GV) := unify_aux(V0,V1,Pa·l)
    if V = TOP or GV = TOP then
         return (TOP,TOP)
    else L := L ∧ l:V
         G := unify(G,GV)
         if G = TOP then return (TOP,TOP)
  return (L,G)
```

Figure 6: The unification procedure

statement.

The most interesting case is the treatment of a pointer. The functional organization of the algorithm does not allow for side effects on remote parts of the top-level formula (nor would this be good programming style), so we had to find a different way to move a subformula to the destination of the pointer. For that reason, we have defined our procedures so that they return *two* results: a *local result* that has to be built into the output formula at the current location (i.e. the path both input formulae are embedded on) and a *global result* that is used to express 'side effects' of the unification. This global result represents a formula that has to be unified with the top-level result in order to find a formula covering all information contained in the input.

This global result is normally set to NIL, but the procedure **move_formula** must of course produce something different. For the time being, we can assume the preliminary definition of **move_formula** in Figure 7, which will be modified in the next subsection. Here, the local result is the pointer (since we want to keep the information about the path equivalence), whereas the global result is a formula containing the subformula to be moved embedded at its new location.

```
move_formula(F,Pfrom,Pto)
                  ↦   (formula,formula)
  return (<Pto>,Pto:F)
```

Figure 7: Movement of a Subformula — Preliminary Version

The function **unify_complex** unifies conjunctions of label-value-pairs by calling **unify_aux** recursively and placing the local results of these unifications at the appropriate locations. Labels that appear only in one argument are built into the output without modification. If any of the recursive unifications fail, a failure has to be announced. The global results from recursive unifications are collected by top-level unification[5]. The third argument of **unify_aux** and **unify_complex** contains the sequence of labels to the actual location. It is not used in this version but is included in preparation of the more sophisticated treatment of pointers described below.

To perform a top-level unification of two formulae, the call to **unify_aux** is repeated in order to unify the local and global results until either the unification fails or the global result is NIL.

Before extending the algorithm to handle disjunction, we will first concentrate on the question how the termination of this repeat-loop can be guaranteed.

5.1 Avoiding Infinite Loops

There are cases where the algorithm in Figure 6 will not terminate if the movement of subformulae is defined as in Figure 7. Consider the unification of $\mathtt{a}:\langle\mathtt{b}\rangle \wedge \mathtt{b}:\langle\mathtt{a}\rangle$ with $\mathtt{a}:\Phi$. Here, the formula Φ

[5] If we allow the global result to be a *list of formulae*, this recursion could be replaced by list-concatenation. However, this would imply modifications in the top-level loop and would slightly complicate the treatment of disjunction.

will be moved along the pointers infinitely often and the repeat-loop in **unify** will never terminate. An algorithm that terminates for arbitrary input must include precautions to avoid the introduction of cyclic pointer chains or it has to recognize such cycles and handle them in a special way.

When working with pointers, the standard technique to avoid cycles is to follow pointer chains to their end and to install a new pointer only to a location that does not yet contain an outgoing pointer. For different reasons, dereferencing is not the method of choice in the context of our treatment of disjunction (see [Eisele 87] for details). However, there are different ways to avoid cyclic movements. A total order '$<_p$' on all possible locations (i.e. all paths) can be defined such that, if we allow movements only from greater to smaller locations, cycles can be avoided. A pointer from a greater to a smaller location in this order will be called a *positive pointer*, a pointer from a smaller to a greater location will be called *negative*. But we have to be careful about chosing the right order; not any order will prevent the algorithm from an infinite loop.

For instance, it would not be adequate to move a formula along a pointer from a location p to its extension $p \cdot q$, since the pointer itself would block the way to its destination. (The equivalence class contains $\langle p \rangle, \langle p\ q \rangle, \langle p\ q\ q \rangle \ldots$ and it makes no sense to choose the last one as a representative). Since cyclic feature structures can be introduced inadvertently and should not lead to an infinite loop in the unification, the first condition the order '$<_p$' has to fulfill is:

$$p <_p pq \quad \text{if} \quad q \neq \varepsilon$$

The order must be defined in a way that positive pointers can not lead to even indirect cycles.

This is guaranteed if the condition

$$p <_p q \quad \Rightarrow \quad rps <_p rqs$$

holds for arbitrary paths p, q, r and s.

We get an order with the required properties if we compare, in the first place, the length of the paths and use a lexicographic order $<_l$ for paths of the same length. A formal statement of this definition is given in Figure 8.

Note that positive pointers can turn into negative ones when the structure containing them is moved, as the following example shows:

$$\underset{pos.}{\texttt{a:b:c:d:}\langle \texttt{a b e}\rangle} \quad \cup \quad \underset{pos.}{\texttt{a:b:c:}\langle \texttt{f}\rangle}$$
$$= \quad \underset{pos.}{\texttt{a:b:c:}\langle \texttt{f}\rangle} \quad \wedge \quad \underset{neg.}{\texttt{f:d:}\langle \texttt{a b e}\rangle}$$

$$p <_p q \quad \text{if} \quad |p| < |q|$$
$$\text{or if} \quad |p| = |q|,\ p = rl_1s,\ q = rl_2t,\ r,s,t \in L^*,\ l_i \in L,\ l_1 <_l l_2$$

Figure 8: An Order on Locations in a Formula

However, we can be pragmatic about this point; the purpose of ordering is the avoidance of cyclic *movements*. Towards this end, we only have to avoid *using* negative pointers, not writing them down.

To avoid movement along a negative pointer, we now make use of the actual location that is provided by the third argument of **unify_aux** and **unify_complex** and as the second argument of **move_formula**.

```
move_formula(F,P_from,P_to)
                    ↦  (formula,formula)
  if P_to <_p P_from then
      return (<P_to>,P_to:F)
  else if P_to = P_from then
      return(F,NIL)
  else return (F,P_to:<P_from>)
```

Figure 9: Movement of a Subformula — Correct Version

The definition of **move_formula** given in Figure 7 has to be replaced by the version given in Figure 9. We distinguish three cases:

- If the pointer is positive we proceed as usual.
- If it points to the actual location, it can be ignored (i.e. treated as NIL). This case occurs, when the same path equivalence is stated more than once in the input.
- If the pointer is negative, it is inverted by installing at its destination a pointer to the actual position.

5.2 Incorporating Disjunction

The procedure **unify_disj** in Figure 10 has four arguments: the formula to unify with the disjunction (which also can be a disjunction), both disjuncts, and the actual location. In the first two statements, the unifications of the formula A with the disjuncts B and C are performed independently. We can distinguish three main cases:

- If one of the unifications fails, the result of the other is returned without modification.
- If both unifications have no global effect *or* if the global effects happen to result in the same

```
unify_disj(A,B,C,P_a)
                . ↦ (formula,formula)
  (L_1,G_1) := unify-aux(A,B,P_a)
  (L_2,G_2) := unify-aux(A,C,P_a)
  if L_1 = TOP or G_1 = TOP then
      return (L_2,G_2)
  else if L_2 = TOP or G_2 = TOP then
      return (L_1,G_1)
  else if G_1 = G_2 then
      return (L_1∨L_2,G_1)
  else return (NIL,pack(unify(P_a:L_1,G_1)∨
                        unify(P_a:L_2,G_2)))
```

Figure 10: Unification with a Disjunction

formula, a disjunction is returned as local result and the common global result of both disjuncts is taken as the global result for the disjunction.

- If both unifications have different global results, we can not return a disjunction as local result, since remote parts of the resulting formula depend on the choice of the disjunct at the actual location. This case arrives if one or both disjuncts have outgoing pointers *and* if one of these pointers has been actually used to move a subformula to its destination.

The last point describes exactly the case where the scope of a disjunction has to be extended to a higher level due to the interaction between disjunction and path equivalence, as was shown in Figure 3. A simple treatment of such effects would be to return a disjunction as global result where the disjuncts are the global results unified with the corresponding local result embedded at the actual position. However, it is not always necessary to return a top-level disjunction in such a situation. If the global effect of a disjunction concerns only locations 'close' to the location of the disjunction, we get two global results that differ only in an embedded substructure. To minimize the 'lifting' of the disjunction, we can assume a procedure **pack** that takes two formulae X and Y and returns a formula equivalent to $X \vee Y$ where the disjunction is embedded at the lowest possible level.

Although the procedure **pack** can be defined in a straightforward manner, we refrain from a formal specification, since the discussion in the next section will show how the same effect can be achieved in a different way.

6 Implementation

We now have given a complete specification of a unification algorithm for formulae in **ENF**. However, there are a couple of modifications that can be applied to it in order to improve its efficiency. The improvements described in this section are all part of our actual implementation.

Unification of Two Pointers

If both arguments are pointers, the algorithm in Figure 6 treats one of them in the same way as an arbitrary formula and tries to move it to the destination of the other pointer. Although this treatment is correct, some of the necessary computations can be avoided if this case is treated in a special way. Both pointer destinations and the actual location should be compared and pointers to the smallest of these three paths should be installed at the other locations.

Special Treatment of Atomic Formulae

In most applications, we do not care about the equivalence of two paths if they lead to the same atom. Under this assumption, when moving an atomic formula along a pointer, the pointer itself can be replaced by the atom without loss of information. This helps to reduce the amount of global information that has to be handled.

Ordering Labels

The unification of conjunctions that contain many labels can be accelerated by keeping the labels sorted according to some order (e.g. $<_l$). This avoids searching one formula for each label that occurs in the other.

Organisation of the Global Results on a Stack

In the algorithm described so far, the global result of a unification is collected, but is – apart from disjunction – not used before the traversal of the input formulae is finished. When formulae containing many pointers are unified, the repeated traversal of the top-level formula slows down the unification, and may lead to the construction of many intermediate results that are discarded later (after having been copied partially).

To improve this aspect of the algorithm, we have chosen a better representation of the global result. Instead of one formula, we represent it as a stack of

formulae where the first element holds information for the actual location and the last element holds information for the top-level formula. Each time a formula has to be moved along a pointer, its destination is compared with the actual location and the common prefix of the paths is discarded. From the remaining part of the actual location we can determine the first element on the stack where this information can be stored. The rest of the destination path indicates how the information has to be represented at that location.

When returning from the recursion, the first element on the stack can be popped and the information in it can be used immediately.

This does not only improve efficiency, but has also an effect on the treatment of disjunction. Instead of trying to push down a top-level disjunction to the lowest possible level, we climb up the stacks returned by the recursive unifications and collect the subformulae until the rests of the stacks are identical. In this way, 'lifting' disjunctions can be limited to the necessary amount without using a function like **pack**.

Practical Experiences

In order to be compatible with existing software, the algorithm has been implemented in PROLOG. It has been extended to the treatment of unification in an LFG framework where indirectly specified labels (e.g in the equation $(\uparrow (\downarrow \text{pcase})) = \downarrow$), set values and various sorts of constraints have to be considered.

This version has been incorporated into an existing grammar development facility for LFGs [Eisele/Dörre 86,Eisele/Schimpf 87] and has not only improved efficiency compared to the former treatment of disjunction by backtracking, but also helps to survey a large number of similar results when the grammar being developed contains (too) much disjunction. One version of this system runs on PCs with reasonable performance.

7 Comparison with Other Approaches

7.1 Asymptotical Complexity

Candidates for a comparison with our algorithm are the naive multiplying-out to DNF, Kasper's representation of general disjunction [Kasper 87b], and Karttunen's treatment of value disjunction [Karttunen 84], also the improved version in [Bear 87]. Since satisfiability of formulae in FML is known to be an NP-complete problem, we cannot expect better than exponential time complexity in the worst case. Nevertheless it might be interesting to find cases where the asymptotic behaviour of the algorithms differ. The following statements – although somewhat vague – may give an impression of strong and weak points of the different methods. For each given statement we have specific examples, but their presentation or proofs would be beyond the scope of this paper.

7.1.1 Space Complexity (Compactness of the Representation)

- When many disjunctions concern different substructures and do not depend on each other, our representation uses exponentially less space than expansion to DNF.
- There are cases where Kasper's representation uses exponentially less space than our representation. This happens when disjunctions interact strongly, but an exponential amount of consistent combinations remain.
- Since Karttunen's method enumerates all consistent combinations when several disjunctions concern the same substructure, but allows for local representation in all other cases, his method seems to have a similar space complexity than ours.

7.1.2 Time Complexity

- There are cases where Kasper's method uses exponentially more time than ours. This happens when disjunctions interact so strongly, that only few consistent combinations remain, but none of the disjunctions can be resolved.
- When disjunctions interact strongly, but an exponential amount of consistent combinations remains, our method needs exponential time. An algorithm using Kasper's representation could do better in some of these cases, since it could find out in polynomial time that each of the disjuncts is used in a consistent combination. However, the actual organisation of Kasper's full consistency check introduces exponential time complexity for different reasons.

7.2 Average Complexity and Conclusion

It is difficult to find clear results when comparing the average complexity of the different methods,

since anything depends on the choice of the examples. However, we can make the following general observation:

All methods have to multiply out disjunctions that are not mutually independent in order to find inconsistencies.

Kasper's and Karttunen's methods discard the results of such computations, whereas our algorithm keeps anything that is computed until a contradiction appears. Thus, our method tends to use more space than the others. On the other hand, since Kasper's and Karttunen's methods 'forget' intermediate results, they are sometimes forced to perform identical computations repeatedly.

As conclusion we can say that our algorithm sacrifies space in order to save time.

8 Further Work

The algorithm or the underlying representation can still be improved or extended in various respects:

General Disjunction

For the time being, when a formula is unified with a disjunction, the information contained in it has to be distributed over all disjuncts. This may involve some unnecessary copying of label-value-pairs in cases where the disjunction does not interact with the information in the formula. (Note, however, that in such cases only the first level of the formula has to be copied.) It seems worthwhile to define a *relaxed* ENF, where a formula $(A \vee B) \wedge C$ is allowed under certain circumstances (e.g. when $(A \vee B)$ and C do not contain common labels) and to investigate whether a unification algorithm based on this relaxed normal form can help to save unnecessary computations.

Functional Uncertainty

The algorithm for unifying formulae with regular path expressions given by Johnson [Johnson 86] gives as a result of a unification a finite disjunction of cases. The algorithm presented here seems to be a good base for an efficient implementation of Johnson's method. The details still have to be worked out.

Acknowledgments

The research reported in this paper was supported by the EUROTRA-D accompanying project (BMFT grant No. 101 3207 0), the ESPRIT project ACORD (P393) and the project LILOG (supported by IBM Deutschland). Much of the inspiration for this work originated from a course about extensions to unification (including the work of Kasper and Rounds) which Hans Uszkoreit held at the University of Stuttgart in spring 1987. We had fruitful discussions with Lauri Karttunen about an early version of this algorithm. Thanks also go to Jürgen Wedekind, Henk Zeevat, Inge Bethke, and Roland Seiffert for helpful discussions and important counterexamples, and to Fiona McKinnon, Stefan Momma, Gert Smolka, and Carin Specht for polishing up our argumentation.

References

[Aït-Kaci/Nasr 86] Aït-Kaci, H. and R. Nasr (1986). LOGIN: A Logic Programming Language with Built-In Inheritance. The Journal of Logic Programming, 1986 (3).

[Bear 87] Bear, J. (1987). Feature-Value Unification with Disjunctions. Ms. SRI International, Stanford, CA.

[Eisele 87] Eisele, A. (1987). Eine Implementierung rekursiver Merkmalstrukturen mit disjunktiven Angaben. Diplomarbeit. Institut f. Informatik, Stuttgart.

[Eisele/Dörre 86] Eisele, A. and J. Dörre (1986). A Lexical Functional Grammar System in Prolog. In: *Proceedings of COLING 1986*, Bonn.

[Eisele/Schimpf 87] Eisele, A. and S. Schimpf (1987). Eine benutzerfreundliche Softwareumgebung zur Entwicklung von LFGen. Studienarbeit. IfI, Stuttgart.

[Gazdar et al. 85] Gazdar, G., E. Klein, G. Pullum and I. Sag (1985). *Generalized Phrase Structure Grammar.* London: Blackwell.

[Johnson 86] Johnson, M. (1986). *Computing with Regular Path Formulas.* Ms. CSLI, Stanford, California.

[Kaplan/Bresnan 82] Kaplan, R. und J. Bresnan (1982). Lexical Functional Grammar: A Formal System for Grammatical Representation. In: J. Bresnan (ed.), *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts.

[Karttunen 84] Karttunen, L. (1984). Features and Values. In: *Proceedings of COLING 1984*, Stanford, CA.

[Kasper 87a] Kasper, R.T. (1987). Feature Structures: A Logical Theory with Application to Language Analysis. Ph.D. Thesis. University of Michigan.

[Kasper 87b] Kasper, R.T. (1987). A Unification Method for Disjunctive Feature Descriptions. In: *Proceedings of the 25th Annual Meeting of the ACL.* Stanford, CA.

[Kasper/Rounds 86] Kasper, R.T. and W. Rounds (1986). A Logical Semantics for Feature Structures. In: *Proceedings of the 24th Annual Meeting of the ACL.* Columbia University, New York, NY.

[Kay 79] Kay, M. (1979). Functional Grammar. In: C. Chiarello et al. (eds.) *Proceedings of the 5th Annual Meeting of the Berkeley Linguistic Society.*

[Kay 85] Kay, M. (1985). Parsing in Functional Unification Grammar. In: D. Dowty, L. Karttunen, and A. Zwicky (eds.) *Natural Language Parsing*, Cambridge, England.

[Smolka/Aït-Kaci 87] Smolka, G. and H. Aït-Kaci (1987). Inheritance Hierarchies: Semantics and Unification. MCC Tech. Rep. No AI-057-87. To appear in: Journal of Symbolic Logic, Special Issue on Unification, 1988.

[Uszkoreit 86] Uszkoreit, H. (1986). Categorial Unification Grammars. In: *Proceedings of COLING 1986*, Bonn.