Disjunctions and Inheritance in the Context Feature Structure System

Martin Böttcher
GMD-IPSI
Dolivostraße 15
D 6100 Darmstadt
Germany
boettche@darmstadt.gmd.de

Abstract

Substantial efforts have been made in order to cope with disjunctions in constraint based grammar formalisms (e.g. [Kasper, 1987; Maxwell and Kaplan, 1991; Dörre and Eisele, 1990].). This paper describes the roles of disjunctions and inheritance in the use of feature structures and their formal semantics. With the notion of contexts we abstract from the graph structure of feature structures and properly define the search space of alternatives. The graph unification algorithm precomputes *nogood* combinations, and a specialized search procedure which we propose here uses them as a controlling factor in order to delay decisions as long as there is no logical necessity for deciding.

1 Introduction

The Context Feature Structure System (CFS) [Böttcher and Könyves-Tóth, 1992] is a unification based system which evaluates feature structures with distributed disjunctions and dynamically definable types for structure inheritance. CFS is currently used to develop and to test a dependency grammar for German in the text analysis project KONTEXT. In this paper disjunctions and inheritance will be investigated with regard to both, their application dimension and their efficient computational treatment.

The unification algorithm of CFS and the concept of virtual agreements for structure sharing has been introduced in [Böttcher and Könyves-Tóth, 1992]. The algorithm handles structure inheritance by structure sharing and constraint sharing which avoids copying of path structures and constraints completely. Disjunctions are evaluated concurrently without backtracking and without combinatoric multiplication of the path structure. For that purpose the path structure is separated from the structure of disjunctions by the introduction of contexts.

Contexts are one of the key concepts for maintaining disjunctions in feature terms. They describe readings of disjunctive feature structures. We define them slightly different from the definitions in [Dörre and Eisele, 1990] and [Backofen *et al.*, 1991], with a technical granularity which is more appropriate for their efficient treatment. The CFS unification algorithm computes a set of *nogood* contexts for all conflicts which occur during unification of structures. An algorithm for contexts which computes from a set of nogoods whether a structure is valid, will be described in this paper. It is a specialized search procedure which avoids the investigation of the full search space of contexts by clustering disjunctions.

We start with some examples how disjunctions and inheritance are used in the CFS environment. Then contexts are formally defined on the basis of the semantics of CFS feature structures. Finally the algorithm computing validity of contexts is outlined.

2 The Use of Disjunctions and Inheritance

Disjunctions

Disjunctions are used to express ambiguity and capability. A first example is provided by the lexicon entry for German *die* (*the*, *that*, ...) in Figure 1. It may be nominative or accusative, and if it is singular the gender has to be feminine.

Those parts of the term which are not inside a disjunction are required in any case. Such parts shall be shared by all "readings" of the term. The internal die :=

$$\left[\begin{array}{ll} \multicolumn{2}{l}{\text{L\_definit-or-relativ@<>}} \\ \text{syn} : & \left[\begin{array}{ll} \text{graph} : \text{die} & \\ \text{categ} : & \left[\begin{array}{l} \text{cas} : \left\{\begin{array}{l}\text{nom}\\ \text{acc}\end{array}\right\} \\ \left\{\begin{array}{l} \text{num} : \text{pl} \\ \left[\begin{array}{l}\text{num} : \text{sg}\\ \text{gen} : \text{fem}\end{array}\right]\end{array}\right\}\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 1: Lexicon Entry for *die*

representation shall provide for mechanisms which prevent from multiplication of independent disjunctions (into dnf).

trans := trans :

$$\left[\begin{array}{l} \left\{\begin{array}{l} \left[\begin{array}{l} \text{dom} : \text{syn} : \text{categ} : \text{gvb} : \text{aktiv} \\ \left\{\begin{array}{l} \left[\text{syn} : \text{categ} : \left[\begin{array}{l}\text{class} : \text{nomn}\\ \text{cas} : \text{acc}\end{array}\right]\right] \\ \left[\begin{array}{l} \text{syn} : \left[\begin{array}{l}\text{lexem} : \text{hypo'}\\ \text{categ} : \text{class} : \text{ssent}\end{array}\right] \\ \text{prn} : \text{none}\end{array}\right]\end{array}\right\} \\ \text{<tree-filler> = <role-filler trans>}\end{array}\right] \\ \left[\begin{array}{l} \text{dom} : \text{syn} : \text{categ} : \left[\begin{array}{l}\text{gvb} : \text{passiv}\\ \text{rel} : \#1\end{array}\right] \\ \text{syn} : \left[\begin{array}{l} \text{categ} : \left[\begin{array}{l}\text{class} : \text{prpo}\\ \text{rel} : \#1\end{array}\right] \\ \text{lexem} : \left\{\begin{array}{l}\text{von}\\ \text{durch}\end{array}\right\}\end{array}\right] \\ \text{<tree-filler> = <role-filler agens>}\end{array}\right]\end{array}\right\} \\ \text{v-verb-trans-slot@<>}\end{array}\right]$$

Figure 2: The Type *trans*

As a second example Figure 2 shows a type describing possible realizations of a transitive object. The outermost disjunction distinguishes whether the dominating predicate is in active or in passive voice. For active predicates either a noun (syn : categ : class : nomn) or a subsentence (syn : categ : class : ssent) is allowed. This way disjunctions describe and restrict the possibility of combinations of constituents.

External Treatment of Disjunctions

The KONTEXT grammar is a lexicalized grammar. This means that the possibility of combinations of constituents is described with the entries in the lexicon rather than in a separated, general grammar. A chart parser is used in order to decide which constituents to combine and maintain the combinations. This means that some of the disjunctions concerning concrete combinations are handled not by the unification formalism, but by the chart. Therefore structure sharing for inheritance which is extensively used by the parser is even more important.

Inheritance

Inheritance is used for two purposes: abstraction in the lexicon and non-destructive combination of chart entries. Figure 3 together with the type *trans* of Figure 2 shows an example of abstraction: The feature structure of ***trans*** is inherited (marked by **@<>**) to the structure for the lexeme ***spielen*** (***to play***) at the destination of the path syn : slots : . A virtual copy of the type structure is inserted. The type ***trans*** will be inherited to all the verbs which allow (or require) a transitive object. It is obvious that it makes sense not only to inherit the structure to all the verbs on the level of grammar description but also to ***share*** the structure in the internal representation, without copying it.

L_spielen :=

$$\left[\begin{array}{ll} \text{syn} : & \left[\begin{array}{ll} \text{lexem} : \text{spielen} & \\ \text{categ} : & \left[\begin{array}{l}\text{fle\_verb} : \text{schwach}\\ \text{pfk} : \text{haben}\end{array}\right] \\ \text{slots} : \text{trans@<>} & \end{array}\right] \\ \multicolumn{2}{l}{\text{v-verb@<>}}\end{array}\right]$$

Figure 3: Lexicon Entry for *spielen*

Inheritance is also extensively used by the parser. It works bottom-up and has to try different combinations of constituents. For single words it just looks up the structures in the lexicon. Then it combines a *slot* of a *functor* with a *filler*. An example is given in Figure 4 which shows a trace of the chart for the sentence *Kinder spielen eine Rolle im Theater.* (Children play a part in the theatre.) In the 6'th block, in the line starting with **...4** the parser combines type **_16** (for the lexicon entry of ***im***) with the type **_17** (for *Theater*) and defines this combination dynamically as type **_18**. **_16** is the functor, **_17** the filler, and **caspn** the name of the slot. The combination is done by unification of feature structures by the CFS system.

The point here is that the parser tries to combine the result **_18** of this step more than once with different other structures, but unification is a destructive operation! So, instead of directly unifying the structures of say **_7** and **_18** (**_11** and **_18**, ...), **_7** and **_18** are inherited into the new structure of **_20**. This way ***virtual copies*** of the structures are produced, and these are unified. It is essential for efficiency that a virtual copy does not mean that the structure of the type has to be copied. The lazy copying approach ([Kogure, 1990], and [Emele, 1991] for lazy copying in TFS with historical backtracking) copies only overlapping parts of the structure. CFS avoids even this by structure- and constraint-sharing.

For common sentences in German, which tend to be rather long, a lot of types will be generated. They supply only a small part of structure themselves (just the path from the functor to the filler and a simple slot-filler combination structure). The bulk of the

```
1: Kinder
     _1 : Kinder                              open/sat

2: spielen
...1 _2 : spielen                             open
     _3 : spielen _2  subje Kinder _1         open/sat
     _4 : spielen _2  trans Kinder _1         open

3: eine
...2 _5 : eine                                open/sat

4: Rolle
...3 _6 : Rolle                               open/sat
...2 _7 : Rolle _6    refer eine _5           open/sat
     _11: spielen _3  trans Rolle _7          open/sat
...1 _14: spielen _2  trans Rolle _7          open

5: im
...4 _16: im                                  open

6: Theater
...5 _17: Theater                             open/sat
...4 _18: im _16      caspn Theater _17 open/sat
...3 _19: Rolle _6    caspp im _18            open/sat
...2 _20: Rolle _7    caspp im _18            open/sat
     _21: spielen _11 caspp im _18            open/sat
...1 _22: spielen _14 caspp im _18            open
     _26: spielen _3  trans Rolle _20         open/sat
...1 _29: spielen _2  trans Rolle _20         open

7: .
...6 _30: .                                   open
     _31: . _30       praed spielen _26 sat
     _32: . _30       praed spielen _21 sat
```

Figure 4: Chart for *Kinder spielen* ...

structure is shared among the lexicon and all the different combinations produced by the parser.

Avoiding Recursive Inheritance

Recursive inheritance would be a means to combine phrases in order to analyze (and generate) without a parser (as in TFS). On the other hand a parser is a controlled device which e.g. knows about important paths in feature structures describing constituents, and which can do steps in a certain sequence, while unification in principle is sequence-invariant. We think that recursion is not in principle impossible in spite of CFS' concurrent treatment of disjunctions, but we draw the borderline between the parser and the unification formalism such that the cases for recursion and iteration are handled by the parser. This seems to be more efficient.

The Connection between Disjunctions and Types

The similarity of the relation between disjunctive structure and disjunct and the relation between type and instance is, that in a set theoretic semantics (see below) the denotation of the former is a superset of the denotation of the latter. The difference is that a disjunctive structure is invalid, i.e. has the empty set as denotation, if each disjunct is invalid. A type, however, stays valid even when all its currently known instances are invalid. This distinction mirrors the uses of the two: inheritance for abstraction, disjunctions for complete enumeration of alternatives. When an external system, like the chart of the parser, keeps track of the relation between types and instances disjunctions might be replaced by inheritance.

3 Contexts and Inheritance

This chapter introduces the syntax and semantics of CFS feature terms, defines contexts, and investigates the relation between type and instance concerning the validity of contexts. We want to define contexts such that they describe a certain reading of a (disjunctive) term, i.e. chooses a disjunct for some or all of the disjunctions. We will define validity of a context such that the intended reading has a non-empty denotation.

The CFS unification algorithm as described in [Böttcher, Könyves-Tóth 92] computes a set of invalid contexts for all unification *conflicts*, which are always conflicts between constraints expressed in the feature term (or in types). The purpose of the definition of contexts is to cover all possible conflicts, and to define an appropriate search space for the search procedure described in the last part of this paper. Therefore our definition of contexts differ from those in [Dörre and Eisele, 1990] or [Backofen *et al.*, 1991].

Syntax and Semantics of Feature Terms

Let $A = \{a, \ldots\}$ be a set of atoms, $F = \{f, f_i, g_i, \ldots\}$ a set of feature names, $D = \{d, \ldots\}$ a set of disjunction names, $X = \{x, y, z, \ldots\}$ a set of type names, $I = \{i, \ldots\}$ a set of instantiation names. The set of terms $T = \{t, t_1, \ldots\}$ is defined by the recursive scheme in Figure 5. A sequence of type definitions is $x := t_1 \;\; y := t_2 \;\; z := t_3 \;\ldots$.

a	atom
$f : t$	feature value pair
$[t_1 \ldots t_n]$	unification
$\{t_1 \ldots t_n\}_d$	disjunction
$<f_1 \ldots f_n> = <g_1 \ldots g_m>$	path equation
$x@<>_i$	type inheritance

Figure 5: The Set of Feature Terms T

The concrete syntax of CFS is richer than this definition. Variables are allowed to express path equations, and types can be unified destructively. Cyclic path equations (e.g. $<> = <g_1 \ldots g_m>$) are supported, but recursive type definition and negation are not supported, yet.

In order to define contexts we define the set of disjunctions of a term, the disjuncts of a disjunction, and deciders as (complete) functions from disjunctions to disjuncts. M_i is a mapping substituting all disjunction names d by $i(d)$, where i is unique for each instantiation.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{dis}: T \rightarrow 2^D,\ \mathit{sub}: D \rightarrow 2^N,} \\
\quad \mathit{dis}(a) & := & \{\} \\
\quad \mathit{dis}(<p> = <q>) & := & \{\} \\
\quad \mathit{dis}(f : t) & := & \mathit{dis}(t) \\
\quad \mathit{dis}(x@<>_i) & := & \mathit{dis}(M_i(t)) | x := t \\
\quad \mathit{dis}([t_1, .., t_n]) & := & \bigcup_j \mathit{dis}(t_j) \\
\quad \mathit{dis}(\{t_1, .., t_n\}_d) & := & \{d\} \cup \bigcup_j \mathit{dis}(t_j), \\
 & & \mathit{sub}(d) := \{1, ..., n\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{deciders}(t) := \\
\quad \{\mathit{choice} : \mathit{dis}(t) \rightarrow N | \mathit{choice}(d) \in \mathit{sub}(d)\}
\end{array}
$$

Figure 6 defines the interpretation $[\![t]\!]_c$ of deciders c w.r.t. terms t as subsets of some universe U (similar to [Smolka, 1988], without sorts, but with named disjunctions and instantiations).

$$
\boxed{
\begin{array}{ll}
\multicolumn{2}{l}{a^I \in U,} \\
\multicolumn{2}{l}{\underline{f^I : U_\perp \rightarrow U}_\perp, f^I(a^I) = \perp, f^I(\perp) = \perp,} \\
[\![a]\!]_c & :=\{a^I\} \\
[\![f : t]\!]_c & :=\{s \in U | f^I(s) \in [\![t]\!]_c\} \\
[\![\,[t_1..t_n]\,]\!]_c & :=\bigcap_i [\![t_1]\!]_c \\
[\![\,\{t_1..t_n\}_d\,]\!]_c & :=[\![t_{c(d)}]\!]_c \\
[\![<f_1..f_n> = <g_1..g_m>]\!]_c & :=\{s \in U | f_n^I(..f_1^I(s)) = \\
 & \qquad g_m^I(..g_1^I(s)) \neq \perp\} \\
[\![x@<>_i]\!]_c & :=\{s \in U | x := t \Rightarrow \\
 & \qquad s \in [\![M_i(t)]\!]_c\}
\end{array}
}
$$

Figure 6: Decider Interpretation

Similar to deciders we define *specializers* as *partial functions* from disjunctions to disjuncts. We also define a partial order $\preceq_t$ on specializers of a term:

$$
\begin{array}{l}
c_1 \preceq_t c_2 \text{ iff} \\
\quad \forall_{d \in \mathit{dis}(t)} \ (c_2 \text{ is defined on } d \wedge c_2(d) = j) \\
\qquad \Rightarrow c_1(d) = j
\end{array}
$$

The interpretation function can be extended to specializers now: If c is a specializer of t, then

$$[\![t]\!]_c := \bigcup_{c' \in \mathit{deciders}(t) \wedge c' \preceq_t c} [\![t]\!]_{c'}$$

A specializer is *valid* iff it's denotation is not empty. For the most general specializer, the function $c_\top$ which is undefined on each disjunction, we get the interpretation of the term:

$$[\![t]\!] := [\![t]\!]_{c_\top}$$

Contexts

Contexts will be objects of computation and representation. They are used in order to record validity for distributed disjunctions. We give our definition first, and a short discussion afterwards.

For the purpose of explanation we restrict the syntax concerning the composition of disjunctions. We say that a disjunctive subterm $\{...\}_d$ of t is *outwards* in t if there is no subterm $\{.., t_j, ..\}_{d'}$ of t with $\{...\}_d$ subterm of t_j. We require for each disjunctive subterm $\{...\}_d$ of t and each subterm $\{.., t_j, ..\}_{d'}$ of t: if $\{...\}_d$ is outwards in t_j then *each* subterm $\{...\}_d$ of t is outwards in t_j. This relation between d' and d we define as $\mathit{subdis}(d', j, d)$. Figure 7 shows the definition of contexts.

$$
\boxed{
\begin{array}{l}
\text{A specializer } c \text{ of } t \text{ is a } \textit{context of } t\text{, iff} \\
\quad \forall d, d' \in \mathit{dis}(t): \\
\qquad (c \text{ is defined on } d \wedge \mathit{subdis}(d', j, d)) \\
\qquad \Rightarrow (c \text{ is defined on } d' \wedge c(d') = j)
\end{array}
}
$$

Figure 7: Definition of Contexts

The set of contexts and a bottom element $\perp$ form a lattice $(\preceq_t, C_{t\perp})$. The infimum operator of this lattice we write as $\sqcap_t$. We drop the index $_t$ from operators whenever it is clear which term is meant.

Discussion: E.g. for the term

$$
\left\{ \begin{array}{l} f : \left\{ \begin{array}{l} t' \\ t'' \end{array} \right\}_{d_2} \\ g : t''' \end{array} \right\}_{d_1}
$$

$(d_1 \rightarrow 2, d_2 \rightarrow 1)$ is a specializer but not a context. We exclude such specializers which have more general specializers $(d_1 \rightarrow 2)$ with the same denotation. For the same term $(d_2 \rightarrow 1)$ is not a context. This makes sense due to the fact that there is no constraint expressed in the term required in $(d_2 \rightarrow 1)$, but e.g. a at the destination of f is required in $(d_1 \rightarrow 1, d_2 \rightarrow 1)$. We will utilize this information about the dependency of disjunctions as it is expressed in our definition of contexts.

In order to show what contexts are used for we define the relation *is required in* (*requi*) of subterms and contexts of t by the recursive scheme:

$$
\begin{array}{lcl}
 & & t \ \mathit{requi} \ c_\top \\
f : t' \ \mathit{requi} \ c & \Rightarrow & t' \ \mathit{requi} \ c \\
x@<>_i \ \mathit{requi} \ c \wedge x := t' & \Rightarrow & M_i(t') \ \mathit{requi} \ c \\
[.., t', ..] \ \mathit{requi} \ c & \Rightarrow & t' \ \mathit{requi} \ c \\
\{.., t_j, ..\}_d \ \mathit{requi} \ c & \Rightarrow & t_j \ \mathit{requi} \begin{pmatrix} d \rightarrow j \\ d' \rightarrow c(d') \end{pmatrix}
\end{array}
$$

The contexts in which some subterms of t are required, we call *input contexts* of t. Each value constraint at the destination of a certain path and each path equation is required in a certain input context.

Example: In

$$
\left[\begin{array}{l} f : \left\{ \begin{array}{l} a \\ b \end{array} \right\}_{d_1} \\ f : \left\{ \begin{array}{l} b \\ e \end{array} \right\}_{d_2} \end{array} \right]
$$

a is required in $(d_1 \rightarrow 1)$ at the destination of f, and e is required in $(d_2 \rightarrow 2)$ at the destination of f, and the conflict is in the infimum context $(d_1 \rightarrow 1) \sqcap (d_2 \rightarrow 2) = (d_1 \rightarrow 1, d_2 \rightarrow 2)$. This way each conflict is always in one context, and any context might be a context of a conflict. So the contexts are defined with the necessary differentiation and without superfluous elements.

We call the contexts of conflicts *nogoods*. It is not a trivial problem to compute the validity of a term or a context from the set of nogoods in the general case. This will be the topic of the last part (4).

Instantiation

If $x := t$ is a type, and x is inherited to some term $x\texttt{@<>}_i$ then for each context c of x there is a corresponding context c' of $x\texttt{@<>}_i$ with the same denotation.

$$[\![x\texttt{@<>}_i]\!]_{c'} = [\![M_i(t)]\!]_{c'} = [\![t]\!]_c$$
$$c' : \mathit{dis}(M_i(t) \rightarrow N, c'(i(d)) = c(d)$$

Therefore each nogood of t also implies that the corresponding context of the instance term $x\texttt{@<>}_i$ has the empty denotation. It is not necessary to detect the conflicts again. The nogoods can be inherited. (In fact they have to because CFS will never compute a conflict twice.)

If the instance is a larger term, the instance usually will be more specific than the type, and there might be conflicts between constraints in the type and constraints in the instance. In this case there are valid contexts of the type with invalid corresponding contexts of the instance. Furthermore the inheritance can occur in the scope of disjunctions of the instance. We summarize this by the definition of *context mapping* m_i in Figure 8.

$x := t,\ c \in \mathit{contexts}(t)$
$t' = ..x\texttt{@<>}_i..,$
$x\texttt{@<>}_i$ is required in $c' \in \mathit{contexts}(t')$
$m_i{:}\mathit{contexts}(t) \rightarrow \mathit{contexts}(t'),$

$$m_i(c) := \begin{pmatrix} i(d) \rightarrow c(d) \\ d' \rightarrow c'(d') \end{pmatrix}$$

Figure 8: Context Mappings

4 Computing Validity

Given a set of nogood contexts, the disjunctions and the subdis-relation of a term, the question is whether the term is valid, i.e. whether it has a non-empty denotation. A nogood context n means that $[\![t]\!]_n = \{\}$. The answer to this question in this section will be an algorithm, which in CFS is run after all conflicts are computed, because an incremental version of the algorithm seems to be more expensive. We start with an example in order to show that simple approaches are not effective.

$$\left[\left\{ \begin{bmatrix} \mathrm{f} : \mathrm{a} \\ \mathrm{g} : \mathrm{a} \end{bmatrix} \begin{bmatrix} \mathrm{f} : \mathrm{b} \\ \mathrm{g} : \mathrm{b} \end{bmatrix} \right\}_{d_1} \left\{ \begin{bmatrix} \mathrm{g} : \mathrm{b} \\ \mathrm{h} : \mathrm{a} \end{bmatrix} \begin{bmatrix} \mathrm{g} : \mathrm{a} \\ \mathrm{h} : \mathrm{b} \end{bmatrix} \right\}_{d_2} \left\{ \begin{bmatrix} \mathrm{h} : \mathrm{b} \\ \mathrm{f} : \mathrm{b} \end{bmatrix} \begin{bmatrix} \mathrm{h} : \mathrm{a} \\ \mathrm{f} : \mathrm{a} \end{bmatrix} \right\}_{d_3} \right]$$

$$\begin{matrix} (d_1 \rightarrow 1, d_2 \rightarrow 1), & (d_1 \rightarrow 2, d_2 \rightarrow 2), \\ (d_2 \rightarrow 1, d_3 \rightarrow 1), & (d_2 \rightarrow 2, d_3 \rightarrow 2), \\ (d_3 \rightarrow 1, d_1 \rightarrow 1), & (d_3 \rightarrow 2, d_1 \rightarrow 2) \end{matrix}$$

Figure 9: Term and Nogood Contexts

For the term in Figure 9 the unification algorithm of CFS computes the shown nogoods. The term is invalid because each decider's denotation is empty. A strategy which looks for similar nogoods and tries to replace them by a more general one will fail. This example shows that it is necessary at least in some cases to look at (a covering of) more specific contexts.

But before we start to describe an algorithm for this purpose we want to explain why the algorithm we describe does a little bit more. It computes all most general invalid contexts from the set of given nogoods. This border of invalid contexts, the computed nogoods, allows us afterwards to test at a low rate whether a context is invalid or not. It is just the test $\exists n \in \mathit{Computed\text{-}Nogoods} : c \preceq_t n$. This test is frequently required during inspection of a result and during output. Moreover nogoods are inherited, and if these nogoods are the most general invalid contexts, computations for instances will be reduced.

The search procedure for the most general invalid contexts starts from the most general context $c_\top$. It descends through the context lattice and modifies the set of nogoods. We give a rough description first and a refinement afterwards:

Recursive procedure n-1

1. if $\exists n \in \mathit{Nogoods} : c \preceq n$ then return 'bad'.
2. select a disjunction d with c undefined on d and such that the specializer $(d \rightarrow j, d' \rightarrow c(d'))$ is a context. if no such disjunction exists, return 'good'.
3. for each $j \in \mathit{sub}(d)$ recursively call n-1 with $(d \rightarrow j, d' \rightarrow c(d'))$.
4. if each call returns 'bad', then replace all $n \in \mathit{Nogoods} : n \preceq c$ by c and return 'bad'.
5. continue with step 2 selecting a different disjunction.

If we replace the fifth step by

5. return 'good'

n-1 will be a test procedure for validity.

n-1 is not be very efficient since it visits contexts more than once and since it descends down to most specific contexts even in cases without nogoods. In order to describe the enhancements we write: c_1 is relevant for c_2, iff $c_1 \sqcap c_2 \neq \bot$.

The algorithm implemented for CFS is based on the following ideas:

(a) select nogoods relevant for c, return 'good' if there are none

(b) specialize c only by disjunctions for which at least some of the relevant nogoods is defined.

(c) order the disjunctions, select in this order in the step 2.-4. cycle.

(d) prevent multiple visits of contexts by different specialization sequences: if the selected disjunction is lower than some disjunction c is defined on, do not select any disjunction in the recursive calls (do step 1 only).

The procedure will be favorably parametrized not only by the context c, but also by the selection of relevant nogoods, which is reduced in each recursive call (because only 'relevant' disjunctions are selected due to enhencement (b)). This makes the procedure stop at depth linear to the number of disjunctions a nogood is defined on. Together with the ordering (c,d) every context which is more general than any nogood is visited once (step 1 visits due to enhencement (d) not counted), because they are candidates for most general nogood contexts. For very few nogoods it might be better to use a different procedure searching 'bottom-up' from the nogoods (as [de Kleer, 1986, second part] proposed for ATMS).

(a) reduces spreading by recognizing contexts without more specific invalid contexts. (b) might be further restricted in some cases: select only such d with $\forall j \in sub(d) : \exists n \in \textit{relevant-nogoods} : n(d) = j$. (b) in fact clusters disjunctions into mutually independent sets of disjunctions. This also ignores disjunctions for which there are currently no nogoods thereby reducing the search space exponentially.

Eliminating Irrelevant Disjunctions

The algorithm implemented in CFS is also capable of a second task: It computes whether disjunctions are no longer relevant. This is the case if either the context in which the disjunctive term is required is invalid, or the contexts of all but one disjunct is invalid.

Why is this an interesting property? There are two reasons: This knowledge reduces the search space of the algorithm computing the border of most general nogoods. And during inheritance neither the disjunction nor the nogoods for such disjunctions need to be inherited. It is most often during inheritance that a disjunction of a type becomes irrelevant in the instance. (Nobody would write down a disjunction which becomes irrelevant in the instance itself.)

Structure- and constraint sharing in CFS makes it necessary to keep this information because contexts of shared constraints in the type are still defined on this disjunction, i.e. the disjunction stays relevant in the type. Let the only valid disjunct of d be k. The information that either the constraint can be ignored ($c(d) \neq k$) or the disjunction can be ignored ($c(d) = k$) is stored with the instantiation. The context mapping for the instantiation filters out either the whole context or the disjunction.

The algorithm is extended in the following way:

4a. if c is an input context of t and d is a disjunction specializing c and the subcontexts are also input contexts, and if all but one specialization delivers 'bad' the disjunction is irrelevant for t. All subdisjunctions of subterms other than the one which is not 'bad' are irrelevant, too.

Consequences

One consequence of the elimination of irrelevant disjunctions during inheritance is, that an efficient implementation of contexts by bitvectors (as proposed in e.g. [de Kleer, 1986]) with a simple shift operation for context mappings will waste a lot of space. Either sparse coding of these bit vectors or a difficult compactifying context mapping is required. The sparse coding are just vectors of pairs of disjunction names and choices. Maybe someone finds a good solution to this problem. Nevertheless the context mapping is not consuming much of the resources, and the elimination of irrelevant disjunctions is worth it.

5 Conclusion

For the tasks outlined in the first part, the efficient treatment of disjunctions and inheritance, we introduced contexts. Contexts have been defined on the basis of a set theoretic semantics for CFS feature structures, such that they describe the space of possible unification conflicts adequately. The unification formalism of CFS computes a set of nogood contexts, from which the algorithm outlined in the third part computes the border of most general nogood contexts, which is also important for inspection and output. Clearly we cannot find a polynomial algorithm for an exponential problem (number of possible nogoods), but by elaborated techniques we can reduce the effort exponentially in order to get usable systems in the practical case.

References

[Backofen *et al.*, 1991] R. Backofen, L. Euler, and G. Görz. Distributed disjunctions for life. In H. Boley and M. M. Richter, editors, *Processing Declarative Knowledge*. Springer, Berlin, 1991.

[Böttcher and Könyves-Tóth, 1992] M. Böttcher and M. Könyves-Tóth. Non-destructive unification of disjunctive feature structures by constraint sharing. In H. Trost and R. Backofen, editors, *Coping with Linguistic Ambiguity in Typed Feature Formalisms, Workshop Notes*, Vienna, 1992. ECAI '92.

[de Kleer, 1986] J. de Kleer. ATMS. *Artificial Intelligence*, 28(2), 1986.

[Dörre and Eisele, 1990] J. Dörre and A. Eisele. Feature logic with disjunctive unification. In *Proceedings of COLING '90*, Helsinki, 1990.

[Emele, 1991] M. C. Emele. Unification with lazy non-redundant copying. In *Proceedings of the 29'th ACL*, Berkeley, 1991.

[Kasper, 1987] R. Kasper. A unification method for disjunctive feature descriptions. In *Proceedings of the 25'th ACL*, Stanford, 1987.

[Kogure, 1990] K. Kogure. Strategic lazy incremental copy graph unification. In *Proceedings of COLING '90*, Helsinki, 1990.

[Maxwell and Kaplan, 1991] J. T. Maxwell and R. M. Kaplan. A method for disjunctive constraint satisfaction. In M. Tomita, editor, *Current Issues in Parsing Technology*. Kluver Academic Publishers, 1991.

[Smolka, 1988] G. Smolka. A feature logic with subsorts. Lilog Report 33, IBM Deutschland, Stuttgart, 1988.