Term Encoding of Typed Feature Structures

Dale Gerdemann
Seminar für Sprachwissenschaft, Universität Tübingen
Kl. Wilhelmstr. 113
72074 Tübingen, Germany
Email: dg@sfs.nphil.uni-tuebingen.de

1 Introduction

I explore in this paper a variety of approaches to Prolog term encoding typed feature structure grammars. As in Carpenter [3], the signature for such grammars consists of a bounded complete partial order of types under subsumption ($\sqsupseteq$) along with a partial function $\mathsf{Approp}\colon \mathsf{Type} \times \mathsf{Feat} \rightharpoonup \mathsf{Type}$. The appropriateness specification is subject to the constraint that feature-value specifications for subtypes are at least as specific as those for supertypes: if $Approp(\mathbf{t}, \textsc{f}) = \mathbf{s}$ and $\mathbf{t}$ subsumes $\mathbf{t}'$, then $Approp(\mathbf{t}', \textsc{f}) = \mathbf{s}'$ for some $\mathbf{s}'$ subsumed by $\mathbf{s}$.[1]

Previous approaches to term encoding of typed feature structures ([1], [2], [7]), have assumed a similar signature plus additional restrictions such as: limitations on multiple inheritance, or exclusion of more specific feature-value declarations on subtypes. The encoding presented here is subject to no such restrictions. The encoding will ensure that every feature structure is well-typed (Carpenter [3]), i.e., for every feature F on a node with type $\mathbf{t}$, the value of this feature must be subsumed by $Approp(\mathbf{t}, \textsc{f})$. And furthermore, the encoding will ensure, as required by HPSG, that every feature structure is extendible to a maximally specific well-type feature structure.[2]

Previous approaches, discussed in §2, have adopted a technique from Mellish [18] [19] in which each type is encoded as an open-ended data structure representing the path taken through the type hierarchy to reach that type. Or, in other words, a type $\mathbf{t}$ is represented as a sequence of types, starting at the most general type below $\top$ and ending at $\mathbf{t}$, in which each consecutive pair consists of supertype followed by immediate subtype. By bundling features together with the types that introduce them, it is then possible to allow the number of features on a type to increase as the type is further instantiated. The disadvantage of this representation, though, is that there is no unique path leading to a multiply-inherited type or any of its subtypes. While a grammar with multiple-inheritance cannot generally be represented in this approach, it is still possible, as explained in §2.1, to compile the multiple inheritance out of the type hierarchy and then term-encode a semantically equivalent grammar. In this approach, multiple inheritance exists as a convenience to the grammar writer, but is not actually used at run time.

[1] Other more complex constraints can be compiled out [13]. So a system that only uses appropriateness conditions is more general than it might first appear.

[2] See Pollard & Sag [20], p. 21

> . . . a feature structure can be taken as a partial description of any of the well-typed (or totally well-typed, or totally well-typed and sort-resolved) feature structures that it subsumes. We choose to eliminate this possible source of confusion by using only totally well-typed sort-resolved feature structures as (total) models of linguistic entities and AVM diagrams (not feature structures) as descriptions.

It follows from this, that for an AVM to describe something, it must be extendible to a totally well-typed sort-resolved (or *type-resolved*) feature structure. As is standard in computational linguistics, I use the term *feature structure* to mean what Pollard and Sag mean by AVM. It turns out that for computational purposes, we will never be interested in Pollard & Sag's notion of a feature structure. For computational purposes, we want more compact feature structures, which can provably be extended to (totally) well-typed and type-resolved feature structures. See [11] [16] for details.

The encoding presented here will in some instances require the introduction of disjunctions in order to ensure satisfiability of feature structures, i.e., to ensure that feature structures are extendible to maximally specific well-typed feature structures.[3] This introduction of disjunctions may, in the worst case, exponentially increase the size of the grammar. Practical experience with HPSG grammars on the Troll system ([9], [15]) has shown, however, that feature structure compaction techniques can be used to keep this increase reasonably small. In §3, I discuss how these techniques can be incorporated into a term unification approach. In §3.1, I discuss the technique of *unextension*, which involves replacing disjunctions of feature structures with maximally specific types on their nodes by smaller disjunctions of feature structures in which the nodes are labelled by more general types. In §3.2, I discuss the use of *unfilling* to remove uninformative feature-value pairs. I show how this technique can be used not only to make feature structures smaller, but also to eliminate disjunctions. Then, in §3.3, I show how the remaining disjunctions can sometimes be efficiently encoded as *distributed* (or *named*) disjunction. Distributed disjunctions have been discussed elsewhere in the literature, but not in the context of term encoded feature structures. Finally in §4, I summarize the approach taken in this paper and discuss directions of future research.

2 Types-as-Paths Encoding

The types-as-paths encoding, introduced by Mellish [18], uses an open-ended data structure representing the path taken through the type hierarchy to reach that type. For simplicity, let us first consider how to represent types that do not take any features, such as the types subsumed by **a** in fig. 1.

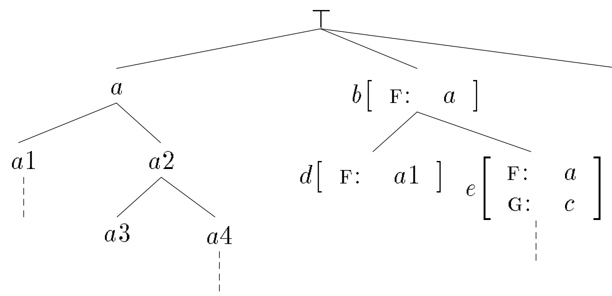

Figure 1: A Simple Type Hierarchy and Appropriateness Specification

type	encoding
a	`a(_)`
$a1$	`a(a1(_))`
$a2$	`a(a2(_))`
$a3$	`a(a2(a3))`
$a4$	`a(a2(a4(_)))`

It is clear that the encodings for **a**, **a2** and **a4** will all unify, whereas the encodings for **a2** and **a1** will not unify.[4] While, in general, the encoding employs open data structures, the example

[3] I am simplifying here for the sake of exposition. The notion of "satisfiable feature structure" is treated in full in King [16].

[4] The encoding used in ALE [4] is similar except that the paths may be gappy. Thus, **a4** could be encoded as any of the following: `_-a4-a2-a`, `_-a4-a`, `_-a4-a2` or `_-a4`. In this example, the path reflects the derivational history of how the **a4** got to be an **a4**. This same principle is used in the implementation of updateable arrays in the Quintus Prolog library. Since the encoding is not unique, a special purpose unifier must be used, which dereferences each type before unifying. Thus this gappy representation is not applicable for the goal of this paper, which is to use ordinary (Prolog-style) term unification.

shows that **a3**, since it is maximally specific, can be encoded as a ground term. If, however, we wish to be able to distinguish in a feature structure between reentrant and non-reentrant instances of **a3**, then we would need encode **a3** also as an open term: `a(a2(a3(_)))`.[5]

This encoding allows a particularly convenient encoding of feature information. If a type introduces n features, then we simply add n additional argument slots to that type in each path. For example, consider the encodings of **b** and **e**:

type	encoding
b	`b(_,_F))`
e	`b(e(_,_G),_F))`

As can be seen from this example, there is essentially no difference between the encoding of a simple type **t** and the encoding of a feature structure of type **t**. The encoding of each type has slots in it where all of the feature value information can be included. The encoding for **e** is particularly instructive. As can be seen, **e** takes two features, which are introduced at two different points along the path. Furthermore, **e** is a subtype of **b**, which has only a single argument position for the feature F. It is clear then that the number of slots for features can increase as a type is further instantiated.

The idea of bundling feature information along with the type that introduces that feature is certainly elegant. However, there is a drawback when not all of the information about a feature is located on a single type. For example, the type **d** inherits a feature F from the supertype **b**, but then adds a more specific value specification for this feature. Consider what happens when the feature structure $b[\text{F} : a]$ (encoded: `b(_,a(_)))`) unifies with the feature structure d (encoded: `b(d,_F))`). The unification of these two encodings is `b(d,a(_)))`, i.e., $d[\text{F} : a]$, which is not well-typed. This, however, is not really a problem specific to the types-as-paths encoding. Any typed feature structure approach with appropriateness specifications needs to incorporate type inferencing. As discussed in §2.2 (see also [11]), one way to maintain appropriateness is to multiply out disjunctive possibilities. For example, $b[\text{F} : a]$ should be multiplied out to: $\{d[\text{F} : a1], e[\text{F} : a]\}$. As seen in the next section, handling multiple inheritance will also involve introducing disjunctions. So the problem of efficiently representing such disjunction (discussed in §3) will be of crucial importance.

2.1 Compiling Out Multiple Inheritance

Multiple inheritance is a genuine problem for the types-as-paths representation. The problem is that if a type can be reached by multiple paths through the type hierarchy, then there is no longer a unique representation for that type. A partial solution to this to use *multi-dimensional inheritance* as in Erbach [7] and Mellish [18]. This idea involves encoding types as a set of paths rather than as a single path. The intuition is supposed to be that each path in this set represents a different dimension in the inheritance hierarchy.

Consider, for example, the commonly used type hierarchy for lists in fig. 2. The types

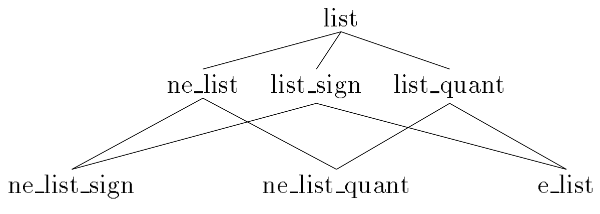

Figure 2: Multiple inheritance in type hierarchy for lists

ne_list, **list_sign** and **list_quant** are not mutually exclusive, so it seems reasonable to represent

[5]This idea is used, for example, in the ProFIT system [7] in order to distinguish between intensional and extensional types (see Carpenter [3] for this distinction). This distinction is certainly important. However, as it is a side issue for this paper, I will ignore it.

these types by a set of paths. **ne_list**, for example, could be represented as `list(ne_list,_,_)` and **ne_list_quant** as `list(ne_list,_,list_quant)`. This seems to work fine except that there would be no way to rule out the unification of the encodings for **ne_list_quant** and **ne_list_sign**. These two types unify to give the non-existent type `list(ne_list,list_sign,list_quant)`. So the restriction on this encoding is the following: if types $\mathbf{t}$ and $\mathbf{t}'$ have a subtype in common, then every subtype of $\mathbf{t}$ must be subsumed by $\mathbf{t}'$. In the **list** example, **ne_list** and **list_sign** have a subtype in common (**ne_list_sign**). However, **list_sign** also subsumes **e_list**, which is not subsumed by **ne_list**. So while this approach may work for some special cases of multiple inheritance, it is not a general solution to the problem.[6]

Given the problems with representing multiple inheritance, it is reasonable to ask how important multiple inheritance is. In fact, given some reasonable closed world assumptions as discussed in the next section, it is possible to compile out all multiple inheritance. To see this, consider what happens when we remove from the list hierarchy the types **list_sign** and **list_quant**. As seen in fig. 3, removing these two types is sufficient to eliminate all multiple

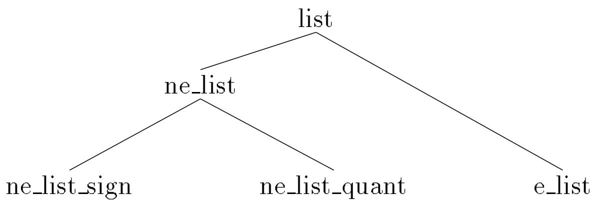

Figure 3: Type hierarchy for lists with multiple inheritance compiled out

inheritance. In order to legitimize removing types from the type hierarchy, two further steps are needed. First, some disjunctive appropriateness specifications must be introduced. For example, for any type $\mathbf{t}$ and feature F with $Approp(\mathbf{t}, \textsc{f}) = \textbf{list\_sign}$, the new appropriateness conditions should include the disjunctive specification $Approp(\mathbf{t}, \textsc{f}) = \{\textbf{ne\_list\_sign}, \textbf{e\_list}\}$.[7] And second, any feature structure containing the type **list_sign** on a node must be compiled into two feature structures: one with **ne_list_sign** and one with **e_list**. So some lexical entries or rules may have to be compiled out into multiple instances. The question of how to deal with this introduced disjunction is treated in §3.

2.2 Closed World Assumptions

So far, we have seen that a closed world interpretation of the type hierarchy will allow us to compile out multiple inheritance at the cost of introducing some disjunctions into the grammar. But the closed world assumption is not a condition that is imposed solely for the purpose of term encoding. As shown in Gerdemann & King [10] [11], the closed world assumption is needed if types are to be used to encode any kind of feature cooccurrence restrictions. As noted by Copestake et al. [5] this deficiency of open-world type systems leads to serious problems for expressing any kind of linguistic constraints.

While the closed world assumption is clearly needed, it is also the case that there is a price to be paid for maintaining this condition. In particular, Gerdemann & King [10] [11] showed that maintaining this condition will sometimes involve multiplying out disjunctive possibilities. For example, consider the type hierarchy in fig. 4. Every feature structure of type **a** must ultimately be resolved to either a feature structure of type $\mathbf{a}'$ or of type $\mathbf{a}''$. So the feature structure $a[\textsc{f}\colon \mathit{bool}, \textsc{g}\colon \mathit{bool}]$ really represents the set of resolvants $\{a'[\textsc{f}\colon +, \textsc{g}\colon -], a''[\textsc{f}\colon -, \textsc{g}\colon +]\}$ and the feature structure $a[\textsc{f}\colon \boxed{1}, \textsc{g}\colon \boxed{1}\,]$ really represents the empty set of feature structures.[8] To make our terminology precise, let us call feature structures such as $a'[\textsc{f}\colon +], a'[\textsc{f}\colon -], a''[\textsc{f}\colon +]$

[6] A second problem with the approach is that it provides no way to attach features to multiply inherited types. So if **ne_list_quant** has some features which are not inherited from either **ne_list** or **list_quant**, then there is no position in either path of types to which these features can be attached.

[7] Such disjunctive appropriateness specifications are allowed, at least internally, in the Troll system [9].

[8] Note that this last feature structure would be considered "well-typed" in ALE. For ALE, one must assume

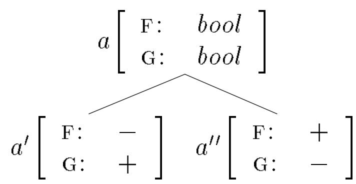

Figure 4: Type hierarchy requiring type resolution

and $a''[\text{F}:\ -]$ *extensions* of the feature structure $a[\text{F}:\ \textit{bool}]$. So a feature structure α can be extended by replacing the type on each node by a species subsumed by that type. The set of resolvants is then the set of extensions which also satisfy the appropriateness conditions: in this case $\{a'[\text{F}:\ +], a''[\text{F}:\ -]\}$. Note that the resolvants of a feature structure need not be totally well typed in the sense of Carptenter [3], i.e., there can be appropriate features such as G which are not in the resolvant.[9] This fact will be of great importance when we consider efficient representations for sets of resolvants in the next section.

So if our term representation for typed feature structures is to maintain the constraints imposed by the appropriateness specification, it looks as if we will again have to expand feature structures out into multiple instances. In fact, as discussed in §3, there are methods both for reducing the need for disjunctions and for compactly representing those disjunctions that can't be eliminated. But before considering these methods, let us first consider one very desirable property of resolved feature structures, namely that the resolved feature structures are closed under unification.

If we use a specialized feature structure unifier, then there exists the possibility of building in type inferencing as part of unification. If all unification is simply term unification, however, then we don't have this option. We need all type inferencing to be static, i.e., applied at compile time.[10] So when two feature structures which satisfy appropriateness conditions unify, the result of this unification must also satisfy appropriateness conditions. As shown in Gerdemann & King [11] and King [16], this is indeed the case for resolved feature structures. Intuitively, the reason for this is quite simple. Type inferencing in a system like ALE has the effect of increasing the specificity of types on certain nodes in a feature structure. If all of these types are already maximally specific, then ALE-style type inferencing could only apply vacuously.

3 Minimizing Disjunctions

We have now seen two instances in which feature structures may need to be multiplied out into disjunctive possibilities. First, this may arise as a result of eliminating multiple inheritance from the type hierarchy. And second, as a result of the type resolution which is needed in order to ensure static typing. Now in this section, I discuss how the need for this disjunction can be reduced by the technique of *unextension* §3.1 and by *unfilling* §3.2. And then in §3.3, I show how the remaining disjunctions can at least sometimes be efficiently represented by using distributed disjunctions.

3.1 Unextension

First, consider the disjunction that is introduced by type resolution. Most of this disjunction can be eliminated by using the technique of *unextension*. Recall that in §2.2, I defined the

an open world semantics in which an object described by $a[\text{F}:\ \boxed{1}, \text{G}:\ \boxed{1}\ ]$ is neither of type $\mathbf{a}'$ nor of type $\mathbf{a}''$.

[9] In fact, in general resolved feature structures cannot be totally well typed since total-well typing creates more nodes in a feature structure which would then need to be resolved and then total-well typed again. The result is an infinite loop [11].

[10] Note, for the sake of comparison, that type inferencing is not static in ALE. The unification of well-typed feature structures is not guaranteed to be well-typed. It is, in fact, not even guaranteed to be well-typable. Thus, the ALE unifier must do type inferencing at run time.

extension of a feature structure α to be a feature structure in which the type on each node of α has been replaced by a species subsumed by that type. Now, let us define the extensions of a set of feature structures S as the set of all extensions of the members of S. And then define the unextension of a set of feature structures S as the minimal cardinality set of feature structures S' such that $extensions(S') = S$.[11] So whenever a feature structure is resolved to a large disjunction of feature structures S, we can normally compact this disjunction back down to a much smaller unextended set S'. Since the extensions of S and S' are the same, it is clear that these two sets of feature structures represent the same information.

As a simple—but extremely commonly occurring—example, consider a feature structure F, all of whose extensions are well-typed. In this case, the set of resolvants of F is exactly the set of extensions of F. So the unextended set of resolvants of F is simply $\{F\}$.[12] So resolving and then unextending F appears at first to accomplish nothing. But, in fact, there is a gain. Initially, with the feature structure F we had no guarantee that static typing would be safe. But with the resolved-unextended set $\{F\}$, we now know that there are no non-well typed extensions, so there will never be a need for run-time type inferencing.

As another example, consider the set of resolvants that HPSG allows for:

head-struc[HEAD-DTR: *sign*, COMP-DTRS: *elist*]

Since the only subtype of **head-struc** for which **elist** is an appropriate value on **comp-dtrs** is **head-comp-struc**, we get the following set of resolvants:

{*head-comp-struc*[HEAD-DTR: *word*, COMP-DTRS: *elist*],
head-comp-struc[HEAD-DTR: *phrase*, COMP-DTRS: *elist*]}.

This two element set corresponds exactly to the set of extensions for the following one element unextended set:

{*head-comp-struc*[HEAD-DTR: *sign*, COMP-DTRS: *elist*]}

So the combination of resolving and unextending simply has the effect of bumping the top-level type from **head-struc** to **head-comp-struc**. This then rules out the malformed feature structure that might have been obtained, for example, by unification with a feature structure of type **head-filler-struc**.

Consider now the disjunctions that are introduced by compiling out multiple inheritance. Recall that this technique involves eliminating intermediate types from the hierarchy. It does not remove any species from the hierarchy. Thus, regardless of whether or not multiple inheritance has been compiled out, the resolvants of a feature structure F will be exactly the same. Since, unextension involves replacing species on nodes with intermediate types, the possibilities for unextending a set of feature structures will be reduced when some intermediate types have been removed. So it turns out that compiling out multiple inheritance actually introduces disjunction in a rather indirect manner.

3.2 Unfilling

Another operation that can be used to reduce the need for disjunction is *unfilling*, which is the reverse of Carpenter's [3] fill operation. In general, the purpose of unfilling is to keep

[11] The normal form result of Götz [12] shows, albeit rather indirectly, that there is a unique unextension for the set of resolvants of a feature structure—assuming there are no unary branches in the type hierarchy. For Götz, unextension is an implicit part of his function for resolving feature structures. I have abstracted unextension out as a separate operation purely for expository reasons. In an actual implementation (such as Troll [9]) it makes more sense to follow Götz's approach since it is not very efficient for the compiler to expand feature structures out into huge sets that then have to be collapsed back down.

[12] There is, however, one complication, namely, if there are unary branches in the type hierarchy, then a feature structure might be resolved and then unextended back to a slightly different, but semantically identical feature structure (see Carpenter [3], chap. 9). For our purposes here, it doesn't matter if the unextension of a set of feature structures is not unique.

feature structures small. If a feature in a feature structure has a value which is no more specific than the appropriateness specification would require, then—assuming no reentrancies would be eliminated and no dangling parts of the feature structure would be created—that feature and its value may be removed. So, in HPSG for example, if the AUX feature in a feature structure of type **verb** has the value **boolean**, then this AUX feature can be removed.

Unfilling is most important, however, when it turns out that eliminating features allows us to further apply unextension to eliminate disjunctions. For example, given the type hierarchy and appropriateness specification in fig. 4 above, the feature structure $a[\text{F}: \ bool, \text{G}: \ bool]$ is resolved to $\{a'[\text{F}: \ +, \text{G}: \ -], a''[\text{F}: \ -, \text{G}: \ +]\}$. However, both of the features F and G have uninformative values, i.e., values which tell us nothing more than we already know from the appropriateness specification. In fact, this is true both in the unresolved feature structure and in each of the resolvants. So these two features can be unfilled to give us the new set of resolvants $\{a', a''\}$. This new set of resolvants can now be unextended to the set $\{a\}$.

There is clearly a great deal more that can be said about unfilling and about the class of unfilled feature structures. The issues, however, are not specific to the problem of term encoding. So I will simply refer the reader to the discussion in Gerdemann & King [11] and Götz [12].

3.3 Distributed Disjunctions

Unextension and unfilling can be used to eliminate quite a lot of disjunctions. Unfortunately, not all disjunctions can be eliminated with these techniques. But still, as noted in Gerdemann & King [10] [11], the remaining disjunctions, have a rather special property. All of the resolvants of a feature structure have the same shape. They differ only in the types labelling the nodes. In a graph-unification based approach, this property can be used to allow the use of a relatively simple version of *distributed* (or *named*) disjunction. This device can be used to push top level disjunctions down to local, interacting disjunctions as in this example:[13]

$$\left\{ \begin{bmatrix} \textit{ne\_list\_sign} \\ \text{HEAD: } \textit{sign} \\ \text{TAIL: } \textit{list\_sign} \end{bmatrix}, \begin{bmatrix} \textit{ne\_list\_quant} \\ \text{HEAD: } \textit{quant} \\ \text{TAIL: } \textit{list\_quant} \end{bmatrix} \right\} \Rightarrow \begin{bmatrix} \langle 1\ \textit{ne\_list\_sign}\ \textit{ne\_list\_quant} \rangle \\ \text{HEAD: } \langle 1\ \textit{sign}\ \textit{quant} \rangle \\ \text{TAIL: } \langle 1\ \textit{list\_sign}\ \textit{list\_quant} \rangle \end{bmatrix}$$

The idea is that if the nth alternative is chosen for a disjunction of a particular name, then then nth alternative has to be chosen uniformly for all other instances of disjunction with the same name.

Distributed disjunctions have been discussed in a fair amount of recent literature ([17], [6], [8], [14]). This version of distributed disjunction, however, is particularly simple since only the type labels are involved. It is not at all difficult to modify a graph unification algorithm in order to handle such disjunctions. Such a modified unification algorithm is used, for example, in the Troll system [9].

For term-represented feature structures, however, it will not be possible to directly encode such distributed disjunctions.[14] So rather than encoding distributed disjunctions in a feature

[13] The example is a little unrealistic since the features HEAD and TAIL could be unfilled. To make the example more realistic, imagine the feature structures embedded in a larger feature structure and imagine that the values of HEAD and TAIL are reentrant with other parts of this feature structure, so that these features could not be unfilled without breaking these reentrancies.

[14] Actually there is an alternative representation for types that would allow a limited amount of direct encoding of distributed disjunctions. Following the basic idea of Mellish [18], one could represent each type as a set of species encoded as a *vset*, where a vset is a term $vset(X_0, X_1, \ldots, X_N)$, with the conditions that:

- $X_0 = 0$
- $X_N = 1$
- $X_i = X_{i-1}$ if the i'th possible element is not in the set

With this *types-as-vsets* representation, some dependencies can be represented as variable sharing across vsets. This idea is elaborated upon in an earlier (and longer) version of the present paper.

structure, we must encode them as definite clause attachments to the feature structure. The idea is fairly simple, to efficiently represent a set of feature structures S, we factor S into first term-represented feature structure, $\sqcup S$, expressing the commonalities across all of the members of S, and second, a set of definite clause attachments expressing all of the allowable further extensions. There is, however, one hitch; namely, how do we know that $\sqcup S$ will be expressible as a Prolog term? If $\sqcup S$ is well typed, then there is no problem. But if $\sqcup S$ contains a node labeled with $\mathbf{t}$ with a with an inappropriate feature F, then the types-as-paths representation simply provides no argument position for this inappropriate feature.

A solution to this problem can be found by imposing the feature introduction condition of Carpenter [3]. This condition requires that for each feature F, there is a unique most general type $Intro(\text{F})$ for which this feature is appropriate. Or, the other way around, if F is appropriate for $\mathbf{t}$ and $\mathbf{t}'$, then it will also be appropriate for $\mathbf{t} \sqcup \mathbf{t}''$. It is straightforward to see, then, that given the feature introduction condition, if the feature structures FS and FS' contain no inappropriate features, then $FS \sqcup FS'$ will also contain no inappropriate features.

As an example, consider again the type hierarchy in fig. 1. Suppose that we want to efficiently represent the set:[15]

$$\{d[\text{F}: a1], e[\text{F}: a]\}$$

The generalization of these two feature structures is then term encodable as follows:

$b[\text{F}: a] \equiv$ `b(X, a(Y))`

Suppose now, that we want to use this feature structure as the lexical entry for the word w. For this simple example, we can encode this with just one definite clause attachment:[16]

```
lex(w, b(X, a(Y))) :-
   p(X, Y).

p(d, a1(_)).
p(e(_), _).
```

One should note here the similarity to distributed disjunctions. The term `b(X, a(Y))` represents the underlying shape of the feature structure and the defining clauses for `p` encode dependencies between types. It is, in fact, rather surprising that this division is even possible. One of the features of the types-as-paths encoding is that the features are bundled together with the types that introduce them. So one might not have expected to see these two types of information unbundled in this manner.

4 Conclusions

Some previous approaches to term encoding of typed feature structures have enforced restrictions against multiple inheritance and against having having more specific feature-value declarations on subtypes. But such restrictions make typing virtually useless for encoding any meaningful constraints. The only restriction imposed in the present approach is the feature introduction condition, which is also imposed in ALE ([4]). In fact, even this minor restriction could be eliminated if we were to allow somewhat less efficient definite clause attachments.

We have seen that in order to enforce constraints encoded in the appropriateness specifications, it will sometimes be necessary to use definite clause attachments to encode disjunctive possibilities. Practical experience with the Troll system suggests that not many such attachments will be needed. Nevertheless, they will arise and will therefore need to be processed

[15] Again, this is an unrealistic example (see footnote 13). One would not normally need distributed disjunctions for such a simple case.

[16] Multiple attachments would correspond to named disjunctions with different names.

efficiently. Certainly options such as delaying these goals or otherwise treating them as constraints can be explored. In fact, one of the main advantages of having a term encoding is that so many options are available from all of the literature on efficient processing of logic programs. So the approach to term encoding presented here should really be viewed as just the first step in the direction of efficient processing of typed feature structure grammars.

References

[1] H. Alshawi, editor. *The Core Language Engine.* MIT Press, Cambridge, Mass, 1991.

[2] H. Alshawi, D.J. Arnold, R. Backofen, D. M. Carter, J. Lindop, K. Netter, J. Tsujii, and H. Uszkoreit. Eurotra 6/1: Rule formalism and virtual machine design study. final report. Technical report, SRI International, Cambridge, 1991. The platform specification for ALEP. Typing in ALEP is explained in a variety of succeeding LRE technical reports.

[3] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge Tracts in Theoretical Computer Science 32. Cambridge University Press, 1992.

[4] Bob Carpenter and Gerald Penn. ALE *The Attribute Logic Engine, User's Guide*, 1994.

[5] Ann Copestake, Antonio Sanfilippo, Ted Briscoe, and Valeria de Paiva. The ACQUILEX LKB: An introduction. In *Inheritance, Defaults, and the Lexicon*, pages 148–163. Cambridge University Press, 1993.

[6] Jochen Dörre and Andreas Eisele. Feature logic with disjunctive unification. In *COLING-90 vol. 2*, pages 100–105, 1990.

[7] Gregor Erbach. ProFIT: Prolog with features, inheritance and templates. In *EACL Proceedings, 7th Annual Meeting*, pages 180–187, 1995.

[8] Dale Gerdemann. *Parsing and Generation of Unification Grammars.* PhD thesis, University of Illinois, 1991. Published as Beckman Institute Cognitive Science Technical Report CS-91-06.

[9] Dale Gerdemann and Thilo Götz. Troll: Type resolution system, user's guide, 1994.

[10] Dale Gerdemann and Paul John King. Typed feature structures for expressing and computationally implementing feature cooccurrence restrictions. In *Proceedings of 4. Fachtagung der Sektion Computerlinguistik der Deutschen Gesellschaft für Sprachwissenschaft*, pages 33–39, 1993.

[11] Dale Gerdemann and Paul John King. The correct and efficient implementation of appropriateness specifications for typed feature structures. In *COLING 94, Proceedings*, pages 956–960, 1994.

[12] Thilo Götz. A normal form for typed feature structures. Master's thesis, Universität Tübingen, 1993.

[13] Thilo Götz and Walt Detmar Meurers. Compiling hpsg type constraints into definite clause programs. In *Proceedings of the Thrirty-Third Annual Meeting of the ACL*, Boston, USA, 1995. ACL.

[14] John Griffith. Optimizing feature structure unification with dependent disjunctions. In John Griffith, Erhard Hinrichs, and Tsuneko Nakazawa, editors, *Topics in Constraint Grammar Formalism for Computational Linguistics: Papers Presented at the Workshop on Grammar Formalism for Natural Language Processing held at ESSLLI-94, Copenhagen.*, number SfS-Report-04-95 in SfS Technical Report. Seminar für Sprachwissenschaft: Universität Tübingen, 1994.

[15] Erhard Hinrichs, Detmar Meurers, and Tsuneko Nakazawa. Partial-vp and split-np topicalization in german – an HPSG anaylsis and it's implementation. Technical Report 58, SFB 340, 1994. Arbeitspapiere des Sonderforschungsbereichs 340, Sprachtheoretische Grundlagen für die Computerlinguistik.

[16] Paul John King. Typed feature structures as descriptions. In *COLING 94, Proceedings*, pages 1250–1254, 1994.

[17] John T. Maxwell III and Ronald M Kaplan. An overview of disjunctive constraint satisfaction. In *Proceedings of International Workshop on Parsing Technologies*, pages 18–27, 1989.

[18] Christopher S. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

[19] Christopher S. Mellish. Term-encodable description spaces. In D.R. Brough, editor, *Logic Programming New Frontiers*, pages 189–207. Intellect, Oxford, 1992.

[20] Carl Pollard and Ivan Sag. *Head Driven Phrase Structure Grammar*. CLSI/University of Chicago Press, Chicago, 1994.