Typing as a means for validating feature structures

Anoop Sarkar and Shuly Wintner
Institute for Research in Cognitive Science
University of Pennsylvania
3401 Walnut St., Suite 400A
Philadelphia, PA 19104
anoop@linc.cis.upenn.edu shuly@linc.cis.upenn.edu

November 22, 1999

Abstract

We present a method for validating the consistency of feature structure specifications by imposing a type discipline. A typed system facilitates a great number of compile-time checks: many possible errors can be detected before the grammar is used for parsing. We have constructed a type signature for an existing broad-coverage grammar of English, and implemented a type inference algorithm that operates on the feature structure specifications in the grammar. The algorithm reports occurrences of incompatibility with the type signature. We have detected a large number of errors in the grammar; four types of errors are described in the paper.

1 Introduction

Feature structures are used by a variety of linguistic formalisms as a means for representing different levels of linguistic information. They are usually associated with more elementary structures (such as phrase-structure rules or trees) to provide an additional dimension for stating linguistic generalizations. A variant of feature structures, *typed* feature structures provide yet another dimension for such generalizations. It is common to assume that typed feature structures have linguistic advantages over untyped ones; and that the former are, in general, more efficient to process. In this paper we show how typing can assist in the validation of untyped feature structure specifications.

The technique we suggest in this paper is incorporated into the XTAG grammar development system, which is based on the Tree-Adjoining Grammar (TAG) formalism (Joshi, Levy, and Takahashi, 1975), extended to include lexicalization (Schabes, Abeillé, and Joshi, 1988) and unification-based feature structures (Vijay-Shanker and Joshi, 1991). Tree Adjoining Languages fall into the class of mildly context-sensitive languages, and as such are more powerful than context free languages. The TAG formalism in general, and lexicalized TAGs in particular, are well-suited for linguistic applications. As first shown by Joshi (1985) and Kroch and Joshi (1987), the properties of TAGs permit one to encapsulate diverse syntactic phenomena in a very natural way.

The XTAG grammar development system makes limited use of feature structures which can be attached to nodes in the trees that make up a grammar. Typically, feature structures in XTAG are flat: nesting of structures is very limited. Furthermore, all feature structures in XTAG are finitely bounded: the maximum size of a feature structure can be statically determined. During parsing, feature structures undergo unification as the trees they are associated with are combined. But unification in XTAG is actually highly limited: since all feature structures are bounded, unification can be thought of as an atomic operation. Feature structure specifications can refer to lexical items, tree families or specific (lexically anchored) trees, and are declared in three different formats and three different files. This organization leaves room for several kinds of errors, inconsistencies and typos in feature structure manipulation: undefined features can be referenced, paths can be assigned undefined values, incompatible features can be equated, etc.

We present a method for validating the consistency of feature structure specifications by imposing a type discipline. A typed system facilitates a great number of compile-time checks: many possible errors can be detected before the grammar is used for parsing. We have constructed a type signature for the XTAG English grammar (The XTAG Research Group, 1998), an existing broad-coverage lexicalized TAG grammar of English. Then, we implemented a type inference algorithm that operates on the feature structure specifications in the grammar. The algorithm reports occurrences of incompatibility with the type signature. We have detected a large number of errors in the grammar; four types of errors are described in the paper.

While the method we suggest was tested on an XTAG grammar, it is in principle applicable to any linguistic formalism that uses untyped feature structures, in particular lexical-functional grammar (Kaplan and Bresnan, 1982).

2 The problem

The organization of an XTAG grammar is such that feature structures are specified in three different components of the grammar:

- a *Tree* database defines feature structures attached to tree *families*
- a *Syn* database defines feature structures attached to lexically anchored trees
- a *Morph* database defines feature structures attached to (possibly inflected) lexical entries.

As an example, consider the verb "seems". This verb can anchor several trees, among which are trees of auxiliary verbs, such as the tree βVvx, depicted in figure 1. This tree is associated with

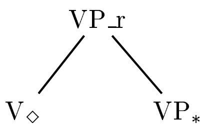

Figure 1: An example tree

the following feature structure descriptions[1] (independently of the word that happens to anchor it):

```
V.t:<agr> = VP_r.b:<agr>
V.t:<assign-case> = VP_r.b:<assign-case>
V.t:<assign-comp> = VP_r.b:<assign-comp>
V.t:<displ-const set1> = VP_r.b:<displ-const set1>
```

[1] We use "handles" such as `V.b` or `NP.t` to refer to the feature structures being specified. Each node in a tree is associated with two feature structures, 'top' (.t) and 'bottom' (.b). Angular brackets delimit feature paths, and slashes denote disjunctive values.

```
V.t:<mainv> = VP_r.b:<mainv>
V.t:<mode> = VP_r.b:<mode>
V.t:<neg> = VP_r.b:<neg>
V.t:<tense> = VP_r.b:<tense>
VP.t:<assign-comp> = ecm
VP.t:<compar> = -
VP.t:<displ-const set1> = -
VP_r.b:<compar> = -
VP_r.b:<conditional> = VP.t:<conditional>
VP_r.b:<perfect> = VP.t:<perfect>
VP_r.b:<progressive> = VP.t:<progressive>
```

When the tree βVvx is anchored by "seems", the lexicon specifies additional constraints on the feature structures in this tree:

```
seem betaVvx VP.b:<mode>=inf/nom, V.b:<mainv> = +
```

Finally, since "seems" is an inflected form, the morphological database specifies more constraints on the node that this word instantiates:

```
seems   seem  V   <agr pers> = 3,
                  <agr num> = sing,
                  <agr 3rdsing> = +,
                  <mode> = ind,
                  <tense> = pres,
                  <assign-comp> = ind_nil/that/rel/if/whether,
                  <assign-case>=nom
```

The actual feature structures that are associated with the lexicalized tree anchored by "seems" are the combination of the three sets of path equations.

This organization leaves room for several kinds of errors, inconsistencies and typos in feature structure manipulation. Nothing in the system can eliminate the following possible errors:

undefined features: Every grammar makes use of a finite set of features in the feature structure specification. However, as the features do not have to be declared, certain bogus features can be introduced unintentionally, either due to typos or because of poor maintenance of the grammar. In a grammar that has an ASSIGN-CASE feature, the following statement is probably erroneous: `V.b:<asign-case> = acc`

undefined values: The same problem can be manifested in values, rather than features. In a grammar where *nom* is a valid value for the ASSIGN-CASE feature, the following statement is probably erroneous: `V.b:<assign-case> = non`

incompatible feature equations: The grammar designer has some notion of what paths can be equated. However, this notion is not formally defined, and so it is possible to find erroneous path equations such as `VP.b:<assign-case> = V.t:<tense>`

Such cases go undetected by XTAG. Their result is always errors in parsing. For example, the statement `V.b:<asign-case> = acc` was presumably supposed to constrain the grammatical derivations to only those in which the ASSIGN-CASE feature had the value *acc*. With the typo, this statement never causes unification to fail (assuming that the feature ASIGN-CASE occurs nowhere else in the grammar); the result is over-generation.

On the other hand, if the statement `V.b:<assign-case> = non` is part of the lexical entry of some verb, and some derivations require that certain verbs have *nom* as their value of ASSIGN-CASE, then that verb would never be a grammatical candidate for those derivations. The result here is under-generation.

3 Introducing typing

The problems discussed above are reminiscent of similar problems in programming languages; in that domain, the solution lies in *typing*: a stricter type discipline provides means for more compile-time checks to be performed, thus tracking potential errors as soon as possible. Fortunately, such a solution is perfectly applicable to the case of feature structures, as typed features structures (TFSs) are well understood (Carpenter, 1992). We briefly survey this concept below.

Typed feature structures are defined over a *signature* consisting of a set of of types (Types) and a set of features (Feats). Types are partially ordered by *subsumption* (denoted '$\sqsubseteq$'). The least upper bound with respect to subsumption of t_1 and t_2 is denoted $t_1 \sqcup t_2$. Each type is associated with a set of *appropriate* features through a function $Aprrop : \text{Types} \times \text{Feats} \rightarrow \text{Types}$. The appropriate values of a feature F in a type t have to be of specified (appropriate) types. Features are inherited by subtypes: whenever F is appropriate for a type t, it is also appropriate for all the types t' such that $t \sqsubseteq t'$. The most general type for which a feature F is appropriate is $Intro(\text{F})$.

Figure 2 graphically depicts a type signature, where greater (more specific) types are presented higher, and the appropriateness specification is displayed above the types. For example, for every feature structure of type *verb* the feature ASSIGN-CASE is appropriate, with values that are at least of type *cases*: $Approp(verb, \text{ASSIGN-CASE}) = cases$.

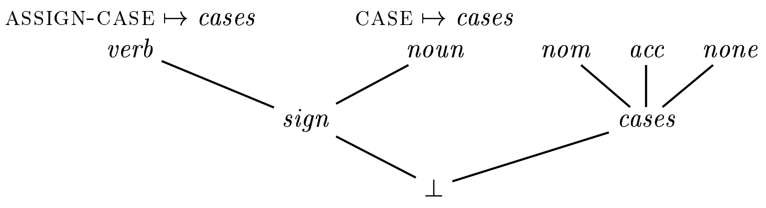

Figure 2: A simple type signature

A formal introduction to the theory of typed feature structures can be found in Carpenter (1992). Informally, a typed feature structure over a signature $\langle \text{Types}, \sqsubseteq, \text{Feats}, Approp \rangle$ differs from an untyped feature structure in two aspects:

- a TFS has a type
- the value of each feature is a TFS – there is no need in atoms in a typed system.

A TFS A whose type it t is *well-typed* iff:

- every feature F in A is such that $Approp(t, \text{F})$ is defined
- every feature F in A has value of type t' such that $Approp(t, \text{F}) \sqsubseteq t'$
- all the substructures of A are well-typed

It is *totally well-typed* if, in addition,

- every feature $\textsc{f}$ such that $Approp(t, \textsc{f})$ is defined occurs in A.

In other words, a TFS is totally well-typed if it has all and only the features that are appropriate for its type, with appropriate values, and the same holds for all its sub-structures.

Totally well-typed TFSs are informative and efficient to process. However, it might be practically difficult for the grammar writer to specify the full information such a structure encodes. To overcome this problem, *type inference* algorithms have been devised than enable a system to automatically *infer* a totally well-typed typed feature structure from a partial description. Partial descriptions can specify:

- the type of a TFS: `V.t:verb`
- a variable, referring to a TFS: `VP.b:assign-case:X`
- a path equation: `VP.b:assign-case = NP.t:case`
- a feature–value pair: `NP.b:case:acc`
- a conjunction of descriptions: `V.t:(sign,assign-case:none)`

The inferred feature structure is the most general TFS which is consistent with the partial description. The inference *fails* iff the description is inconsistent (i.e., describes no feature structure). See figure 3 for some examples of partial descriptions and the TFSs they induce, based on the signature of figure 2.

- `V.t:verb`

$$\texttt{V.t} = \begin{bmatrix} \textit{verb} & \\ \textsc{assign-case}: & \begin{bmatrix}\textit{cases}\end{bmatrix} \end{bmatrix}$$

- `VP.b:assign-case:X`

$$\texttt{VP.b} = \begin{bmatrix} \textit{verb} & \\ \textsc{assign-case}: & \begin{bmatrix}\textit{cases}\end{bmatrix} \end{bmatrix}$$

- `VP.b:assign-case = NP.t:case`

$$\begin{array}{cc} \texttt{VP.b} & \texttt{NP.t} \\ \begin{bmatrix} \textit{verb} & \\ \textsc{assign-case}: & \boxed{1}\begin{bmatrix}\textit{cases}\end{bmatrix} \end{bmatrix} & \begin{bmatrix} \textit{noun} & \\ \textsc{case}: & \boxed{1} \end{bmatrix} \end{array}$$

- `NP.b:case:acc`

$$\texttt{NP.b} = \begin{bmatrix} \textit{noun} & \\ \textsc{case}: & \begin{bmatrix}\textit{acc}\end{bmatrix} \end{bmatrix}$$

- `V.t:(sign,assign-case:none)`

$$\texttt{V.t} = \begin{bmatrix} \textit{verb} & \\ \textsc{assign-case}: & \begin{bmatrix}\textit{none}\end{bmatrix} \end{bmatrix}$$

Figure 3: Inferred TFSs

4 Implementation

In order to validate feature structure specifications in XTAG we have implemented the type inference algorithm suggested by Carpenter (1992, chapter 6). We have manually constructed a type signature suitable for the current use of feature structures in the XTAG grammar of English (The XTAG Research Group, 1998). Then, we applied the type inference algorithm to all the feature structure specifications of the grammar, such that each feature structure was expanded with respect to the signature. We briefly discuss this process below.

4.1 Type inference

The type-inference algorithm was implemented in C, augmented by Lex and Yacc; this is an adaptation of the implementation used in another system, AMALIA, which includes a compiler for unification grammars based on typed feature structures (Wintner and Francez, 1999). The algorithm is schematically depicted in figure 4.

Given a description of a feature structure (such as the ones exemplified in figure 3), the program attempts to construct the most general TFS that satisfies the description. At the first stage, a graph representation of a TFS satisfying the partial description is constructed. Then, the function `infer` is called (see figure 4). In the algorithm, we abuse '*fs*' to denote both a TFS and its root node. The `infer` procedure calls `infer1` and, if successful, `fill`. We take care of possible reentrancies (and, in principle, cycles) in the feature structure by marking nodes as "visited", so that they are not processed more than once by any of the procedures.

Given a typed feature structure *fs* of type t, `infer1` checks all the outgoing edges of *fs*. First, for every such edge labeled f, $Intro(f)$ is computed. The result is unified with t: this ensures that the type of *fs* is the most general type that is appropriate for all the features that occur in *fs*. Note that this step can fail: if *fs* has a feature f such that t and $Intro(f)$ are inconsistent, the unification fails. This rules out occurrences of inappropriate features.

The next step guarantees that the value of each feature f in *fs* is appropriate. If t_1 is the actual type of the node which is the value of an f-labeled edge, and t_2 is the appropriate value for the value of f, then t_1 and t_2 must be consistent – otherwise, a failure is signaled.

Finally, the last step in `infer1` calls this procedure recursively, for each substructure of *fs*. This guarantees that substructures are well-typed as well.

The third procedure, `fill`, simply adds features whenever a TFS is not total. For each added feature, the most general TFS of the appropriate type is constructed as the feature's value. This step can never fail.

From the description above it should be clear how type inference detects inconsistencies of the kind described in section 2. Undefined features and values are immediately detected by the procedures that construct the graph representation of TFSs. Then, if an inappropriate feature is used in some TFS, `infer1` would signal an error in its first step. If an inappropriate value is used for some feature, an error will be signaled in the second step of `infer1`. Finally, if two "unrelated" paths are equated in a description, type inference will assign a type for the end of each paths, and these types are likely to be inconsistent; their unification (by `infer1`) will fail and an error will be signaled.

Type inference is applied off-line, before the grammar is used for parsing. As is the case with other off-line applications, efficiency is not a critical issue. However, it is worth noting that for the grammar we checked (in which, admittedly, feature structures are flat and relatively small), the validation procedure is highly efficient. As a benchmark, we checked the consistency of 1000 trees, each consisting of two to fourteen nodes. The input file, whose size approached 1MB, contained over 33000 path equations. Validating the consistency of the benchmark trees took less than 33 seconds – more than a thousand path equations per second.

```
infer (fs): boolean
    mark all nodes in fs as unvisited;
    if not (infer1(fs)) return FALSE;
    mark all nodes in fs as unvisited;
    fill(fs);
    return TRUE;

infer1 (fs): boolean
    if fs is marked as visited, return TRUE;
    mark fs as visited;
    let t be the type of fs;
    for each feature f occuring in fs,
        let t' be t ⊔ Intro(f);
        if t' = ⊤, return FALSE; else, set t to t';
    set the type of fs to t;
    for each feature f occuring in fs,
        let dtr be the value of f in fs;
        let t1 be the type of dtr;
        let t2 be Approp(t, f);
        if t2 ⋢ t1,
            let t' be t1 ⊔ t2;
            if t' = ⊤, return FALSE; else set the type of dtr to t';
    for each feature f occuring in fs,
        let dtr be the value of f in fs;
        if not infer1(dtr), return FALSE;
    return TRUE;

fill (fs): void
    if fs is marked as visited, return TRUE;
    mark fs as visited;
    let t be the type of fs;
    for each feature f not occuring in fs such that r = Approp(t, f) is defined,
        add the feature f to fs with a single node of type r as its value;
    for each feature f occuring in fs,
        let dtr be the value of f in fs; fill(dtr);
```

Figure 4: Type inference algorithm

4.2 The signature

The signature for the English grammar was constructed manually, by observing the use of feature equations in the grammar and consulting its documentation (The XTAG Research Group, 1998). As noted above, most feature structures used in the grammar are flat, but the number of features in the topmost level is relatively high. Using the meta-language of the ALE system (Carpenter and Penn, 1994), the constructed signature is listed in figure 5.

```
bot sub [sign,agrs,persons,gens,nums,cases,bool,comps,constituents,modes,sel-modes,tenses,conjs,rel-prons,puncts,
         bals,containss,structs,terms].
  sign sub [n_or_v,p_or_n,p_or_v_or_comp,n_or_comp]
       intro [wh:bool,assign-comp:comps,rel-pron:rel-prons,trace:bot,equiv:bool,compar:bool,super:bool,neg:bool].
    n_or_v sub [noun,verb] intro [agr:agrs,conj:conjs,control:bot,punct:puncts,compar:bool,displ-const:constituents].
    p_or_n sub [p,noun].
    p_or_v_or_comp sub [p_or_v,v_or_comp] intro [assign-case:cases].
    p_or_v sub [p,verb] intro [assign-case:cases].
    v_or_comp sub[verb,comp] intro [comp:comps].
    n_or_comp sub[noun,comp] intro [select-mode:sel-modes].
      verb sub [s] intro [inv:bool,mainv:bool,assign-comp:comps,tense:tenses,invlink:bot,mode:modes,
                          passive:bool,conditional:bool,perfect:bool,progressive:bool,contr:bool].
      noun sub [] intro [case:cases,definite:bool,const:bool,rel-clause:bool,pron:bool,quan:bool,card:bool,
                         decreas:bool,gerund:bool,refl:bool,gen:gens,predet:bot,compl:bool].
      p sub [].
      comp sub [] intro [select-mode:sel-modes].
      s sub [] intro [extracted:bool,disc-conj:bool].
  agrs sub [] intro [num:nums,pers:persons,3rdsing:bool].
  persons sub [1,2,3].
    1 sub [].  2 sub [].  3 sub [].
  gens sub [fem,masc,neuter].
    fem sub [].  masc sub [].  neuter sub [].
  nums sub [plur,sing].
    plur sub []. sing sub [].
  cases sub [nom,acc,gen,none].
    nom sub [].  acc sub [].  gen sub [].  none sub [].
  bool sub [+,-].
    + sub []. - sub [].
  comps sub [that,whether,if,for,ecm,rel,inf_nil,ind_nil,ppart_nil,none,nil].
    that sub []. whether sub []. if sub []. for sub []. ecm sub [].
    rel sub []. inf_nil sub []. ind_nil sub []. ppart_nil sub []. nil sub [].
  constituents sub [] intro [set1:bool].
  modes sub [base,ind_inf_ppart_ger,imp,nom,prep,subjunt].
    base sub []. imp sub []. prep sub []. subjunt sub [].
    ind_inf_ppart_ger sub [ind_or_inf,ppart_or_ger].
      ind_or_inf sub [ind,inf].
        ind sub []. inf sub [].
      ppart_or_ger sub [ppart,ger].
        ppart sub []. ger sub [].
  sel-modes sub [ind_inf_ppart_ger].
  tenses sub [pres,past].
    pres sub []. past sub [].
  conjs sub [and,or,but,to,disc,scolon_or_comma_or_nil].
    scolon_or_comma_or_nil sub [scolon,comma,nil]. and sub []. or sub []. but sub []. comma sub [].
    scolon sub []. to sub []. disc sub [].
  puncts sub [] intro [bal:bals,contains:containss,struct:structs,term:terms].
    bals sub [dquote,squote,paren,nil].
      dquote sub []. squote sub []. paren sub [].
    containss sub [] intro [colon:bool,dash:bool,dquote:bool,scolon:bool,squote:bool].
    structs sub [dash,colon,scolon_or_comma_or_nil].
      dash sub []. colon sub [].
    terms sub [per,qmark,excl,nil].
      per sub []. qmark sub []. excl sub [].
 rel-prons sub [ppart_or_ger,adj-clause].
    adj-clause sub [].
```

Figure 5: The underlying signature of the XTAG English grammar

4.3 Results

Applying the type inference algorithm to the XTAG English grammar, we have validated the consistency of all feature structures specified in the grammar. We have been able to detect a great number of errors, which we discuss in this section. The errors can be classified to four different types: ambiguous names; typos; undocumented features; and plain errors.

Ambiguous names This is an obvious error, but one that is not easy to track without the typing mechanism that we discuss in this paper. As the grammar has been developed by as many as a dozen developers, during a period of more than a decade, such errors are probably unavoidable. Specifically, a single name is used for two different features or values, with completely different intentions in mind. We have found several such errors in the grammar.

The feature GEN was used for two purposes: in nouns, it referred to the GENDER, and took values such as *masc, fem* or *neuter*; in pronouns, it was a boolean feature denoting genitive case. We even found a few cases where the values of these incompatible features were equated. As another example, the value *nom* was used to denote both nominative case, where it was an appropriate value for the CASE feature; and to denote a nominal predicate, where it was the appropriate value of the MODE feature. Of course, these two feature have nothing to do with each other and should never be equated (hence, should never have the same value). Finally, values such as *nil* or *none* were used abundantly for a variety of purposes.

Typos Another type of errors that are very difficult to track otherwise are plain typos. The best example is probably a feature that occured about 80 percent of the time as RELPRON and the rest of the time as REL-PRON:

```
S_r.t:<relpron> = NP_w.t:<rel-pron>
```

Undocumented features We have encountered a great number of features and values which were not mentioned in the technical report documenting the grammar. Some of them turned out to be remnants of old analyses that were obsolete; others indicated a need in better documentation.

Some features that were omitted from the grammar due to the validation procedure include PREDET, DISC-CONJ, 3RDSING, among others. Of course, the less features the grammar is using, the more efficient unification (and, hence, parsing) becomes.

Other cases necessitated updates of the grammar documentation. For example, the feature DISPL-CONST was documented as taking boolean values, but turned out to be a complex feature, with a substructure under the feature SET1. The feature GEN (in its gender use) was defined at the top level of nouns, whereas it should have been under the AGR feature.

Other errors Finally, some errors are plain mistakes of the grammar designer. For example, the specification

```
S_r.t:<assign-case> = NP_w.t:<assign-case>
```

implies that ASSIGN-CASE is appropriate for nouns, which is of course wrong; the specification

```
S_r.t:<case> = nom
```

implies that sentences have CASE;

```
V.t:<refl> = V_r.b:<refl>
```

implies that verbs can be REFLexive; etc. A slightly more complex case is triggered by the specification

```
D_r.b:<punct bal> = Punct_1.t:<punct>
```

This should have been either

```
D_r.b:<punct> = Punct_1.t:<punct>
```

or

```
D_r.b:<punct bal> = Punct_1.t:<punct bal>
```

4.4 Additional advantages

Since the feature structure validation procedure practically expands path equations to (most general) totally well-typed feature structures, we have implemented a mode in which the system outputs the expanded TFSs. Users can thus have a better idea of what feature structures are associated with tree nodes, both because all the features are present, and because typing adds information that was unavailable in the untyped specification.

As an example, consider the following specification:

```
PP.b:<wh> = NP.b:<wh>
PP.b:<assign-case> = nom
PP.b:<assign-case> = N.t:<case>
NP.b:<agr> = N.t:<agr>
NP.b:<case> = N.t:<case>
N.t:<case> = nom/acc
```

When it is expanded by the system, the following TFS is output for `PP.b`:

```
PP.b:
[52]p_or_v_or_comp(
  wh:[184]bool,
  assign-comp:[54]comps,
  rel-pron:[55]rel-prons,
  trace:[56]bot,
  equiv:[57]bool,
  compar:[58]bool,
  super:[59]bool,
  neg:[60]bool,
  assign-case:[304]nom)
```

Note that the type of this TFS was set to *p_or_v_or_comp*, indicating that there is not sufficient information for the type inference procedure to distinguish between these three types. Many features that are not explicitly mentioned are added by the inference procedure, with their “default” (most general) values.

The node `N.t` is associated with a TFS, parts of which are:

```
[289]noun(
  wh:[290]bool,
  agr:[298]agrs(
    num:[118]nums,
    pers:[119]persons),
  conj:[299]conjs,
  control:[300]bot,
```

```
displ-const:[302]constituents(
  set1:[153]bool),
case:[304],
definite:[305]bool,
const:[306]bool,
rel-clause:[307]bool,
pron:[308]bool,
quan:[309]bool,
card:[310]bool,
decreas:[311]bool,
gerund:[312]bool,
refl:[313]bool,
gen:[314]gens,
compl:[316]bool)
```

It is worth noting that the type of this TFS was correctly inferred to be *noun*; and that the CASE feature is reentrant with the ASSIGN-CASE feature of the `PP.b` node (through the reentrancy tag [304]), thus restricting it to *nom*, although the specification listed a disjunctive value, *nom/acc*.

5 Further research

We have described in this paper a method for validating the consistency of feature structure specifications in grammars that incorporate untyped feature structures. While the use of feature structures in XTAG is very limited, especially due to the fact that all feature structures are finitely bounded, the method we describe is applicable to feature structure based grammatical formalisms in general; in particular, it will be interesting to test it on broad coverage grammars that are based on unbounded feature structures, such as LFG grammars. Unfortunately, we did not have access to such grammars, but such an application is definitely feasible.

We have applied type inference only statically; feature structures that are created at parse-time are not validated. However, modifying the unification algorithm currently used in XTAG, it is possible to use TFSs in the grammar and apply type inference at run-time. This will enable detection of more errors at run-time; provide for better representation of feature structures; and possibly for more efficient unifications. In a new implementation of XTAG, currently under development (Sarkar, 1999), feature structure specifications are not evaluated as structures are being constructed; rather, they are deferred to the final stage of processing, when only valid trees remain. We plan to apply type inference to the resulting feature structures in this implementation, so that run-time errors can be detected as well.

References

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

Carpenter, Bob and Gerald Penn. 1994. ALE 2.0 user's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 15213, December.

Joshi, Aravind K. 1985. Tree Adjoining Grammars: How much context Sensitivity is required to provide a reasonable structural description. In D. Dowty, I. Karttunen, and A. Zwicky, editors, *Natural Language Parsing*. Cambridge University Press, Cambridge, U.K., pages 206–250.

Joshi, Aravind K., L. Levy, and M. Takahashi. 1975. Tree Adjunct Grammars. *Journal of Computer and System Sciences.*

Kaplan, Ronald and Joan Bresnan. 1982. Lexical functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Mass., pages 173–281.

Kroch, Anthony S. and Aravind K. Joshi. 1987. Analyzing Extraposition in a Tree Adjoining Grammar. In G. Huck and A. Ojeda, editors, *Discontinuous Constituents, Syntax and Semantics*, volume 20. Academic Press.

Sarkar, Anoop. 1999. Combining structural and statistical information: Relevance for efficient processing. Presented at the Seminar on Efficient Processing with High-Level Grammatical Formalisms, Schloss Dagstuhl, Germany, October.

Schabes, Yves, Anne Abeillé, and Aravind K. Joshi. 1988. Parsing strategies with 'lexicalized' grammars: Application to Tree Ad joining Grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, August.

The XTAG Research Group. 1998. A lexicalized tree adjoining grammar for English. IRCS Report 98–18, Institue for Research in Cognitive Science, University of Pennsylvania, 3401 Walnut St, Suite 400A, Philadelphia, PA 19104, August.

Vijay-Shanker, K. and Aravind K. Joshi. 1991. Unification Based Tree Adjoining Grammars. In J. Wedekind, editor, *Unification-based Grammars.* MIT Press, Cambridge, Massachusetts.

Wintner, Shuly and Nissim Francez. 1999. Efficient implementation of unification-based grammars. *Journal of Language and Computation*, 1(1):53–92, April.