Squibs and Discussions

A Note on Typing Feature Structures

Shuly Wintner*
University of Haifa

Anoop Sarkar†
University of Pennsylvania

Feature structures are used to convey linguistic information in a variety of linguistic formalisms. Various definitions of feature structures exist; one dimension of variation is typing: unlike untyped feature structures, typed ones associate a type with every structure and impose appropriateness constraints on the occurrences of features and on the values that they take. This work demonstrates the benefits that typing can carry even for linguistic formalisms that use untyped feature structures. We present a method for validating the consistency of (untyped) feature structure specifications by imposing a type discipline. This method 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 and reports incompatibilities with the signature. We have detected a large number of errors in the grammar, some of which are described in the article.

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 sometimes assumed that typed feature structures have linguistic advantages over untyped ones and that they are, in general, more efficient to process. In this article we show how typing can be useful also for systems that manipulate untyped feature structures.

We present a method for validating the consistency of feature structure specifications by imposing a type discipline. This method facilitates a great number of compile-time checks: many possible errors can be detected before the grammar is used for parsing. Typed systems are used in one linguistic theory, Head-Driven Phrase Structure Grammar (HPSG) (Pollard and Sag 1994), and we present here a different application of them for theories that employ untyped feature structures. We constructed a type signature for the XTAG English grammar (XTAG Research Group 2001), an existing broad-coverage 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 article.

The technique we propose was incorporated into the XTAG grammar development system, which is based on the tree-adjoining grammar (TAG) formalism (Joshi, Levy,

* Department of Computer Science, University of Haifa, Mount Carmel, 31905 Haifa, Israel. E-mail: shuly@cs.haifa.ac.il
† IRCS, University of Pennsylvania, 3401 Walnut Street, Philadelphia, PA-19104. E-mail: anoop@linc.cis.upenn.edu

and Takahashi 1975), lexicalized (Schabes, Abeillé, and Joshi 1988) and augmented by 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 that 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 viewed as an atomic operation. Although the method we propose was tested on an XTAG grammar, it is applicable in principle to any linguistic formalism that uses untyped feature structures, in particular, to lexical-functional grammar (Kaplan and Bresnan 1982).

2. The Problem

XTAG is organized 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; and 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, which is common to all auxiliary verbs, is associated with the feature structure descriptions listed in Figure 1 (independently of the word that happens to anchor it).[1] When the tree βVvx is anchored by *seems*, the lexicon specifies additional constraints on the feature structures in this tree:

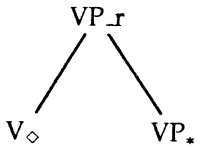
```
seem betaVvx VP.b:<mode> = inf/nom,
             V.b:<mainv> = +
```

Finally, since "seems" is an inflected form, the morphological database specifies more constraints on the node that this word instantiates, as shown in Figure 2.

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. As the features do not have to be declared, however, certain bogus features can be introduced unintentionally, either through typos or because of poor maintenance. In a grammar that has an ASSIGN-CASE feature, the following statement is probably erroneous: `V.b:<asign-case> = acc`.

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) (Vijay-Shanker and Joshi 1991; XTAG Research Group 2001). Angular brackets delimit feature paths, and slashes denote disjunctive (atomic) values.




```
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>
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>
```

Figure 1
An example tree and its associated feature structure descriptions.

```
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
```

Figure 2
The morphological database entry for *seems*.

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 a notion of what paths can be equated, but this notion is not formally defined. Thus, it is possible to find erroneous path equations such as `VP.b:<assign-case> = V.t:<tense>`.

Such cases go undetected by XTAG and result in parsing errors. For example, the statement `V.b:<asign-case> = acc` was presumably supposed to constrain the grammatical derivations to 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 overgeneration.

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 undergeneration.

Note that nothing in the above description hinges on the particular linguistic formalism or its implementation. The same problems are likely to occur in every system that manipulates untyped feature structures.[2]

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 feature structures (TFSs) are well understood (Carpenter 1992). We briefly survey this concept below.

TFSs 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 $Approp$: TYPES $\times$ FEATS $\rightarrow$ 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 $t \sqsubseteq t'$. Each feature F has to be introduced by some most general type *Intro*(F) (and be appropriate for all its subtypes).

Figure 3 graphically depicts a type signature in which 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*: $Approp(verb, \text{ASSIGN-CASE}) = cases$.

A formal introduction to the theory of TFSs is given by Carpenter (1992). Informally, a TFS 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; and the value of each feature is a TFS—there is no need for atoms in a typed system. A TFS A whose type is 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 $Approp(t, \text{F}) \sqsubseteq t'$; and all the substructures of A are well-typed. It is **totally well-typed** if, in addition, every feature F such that $Approp(t, \text{F})$ 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 substructures.

Totally well-typed TFSs are informative and efficient to process. It might be practically difficult, however, for the writer of a grammar to specify the full information such a structure encodes. To overcome this problem, **type inference** algorithms have been devised that enable a system to infer a totally well-typed TFS automatically 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`

2 Some systems could have elaborate mechanisms implemented to deal with each kind of error mentioned here. But typing provides a single mechanism that handles several different kinds of errors simultaneously.

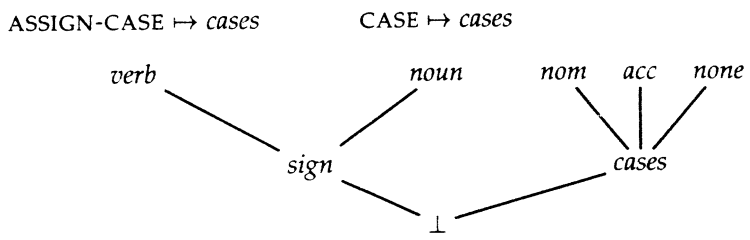


Figure 3
A simple type signature.

- 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 that is consistent with the partial description. The inference **fails** iff the description is inconsistent (i.e., describes no feature structure). See Figure 4 for some examples of partial descriptions and the TFSs they induce, based on the signature of Figure 3.

4. Implementation

To validate feature structure specifications in XTAG we have implemented the type inference algorithm suggested by Carpenter (1992, chapter 6). We manually constructed a type signature suitable for the current use of feature structures in the XTAG grammar of English (XTAG Research Group 2001). 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.

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. It is worth noting, however, 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 1,000 trees, each consisting of two to fourteen nodes. The input file, whose size approached 1MB, contained over 33,000 path equations. Validating the consistency of the benchmark trees took less than 33 seconds (more than a thousand path equations per second).

4.1 The Signature

The signature for the XTAG grammar was constructed manually, by observing the use of feature equations in the grammar and consulting its documentation. As noted above, most feature structures used in the grammar are flat, but the number of features in the top level is relatively high. The signature consists of 58 types and 56 features, and its construction took a few hours. In principle, it should be possible to construct signatures for untyped feature structures automatically, but such signatures will of course be less readable than manually constructed ones.

4.2 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

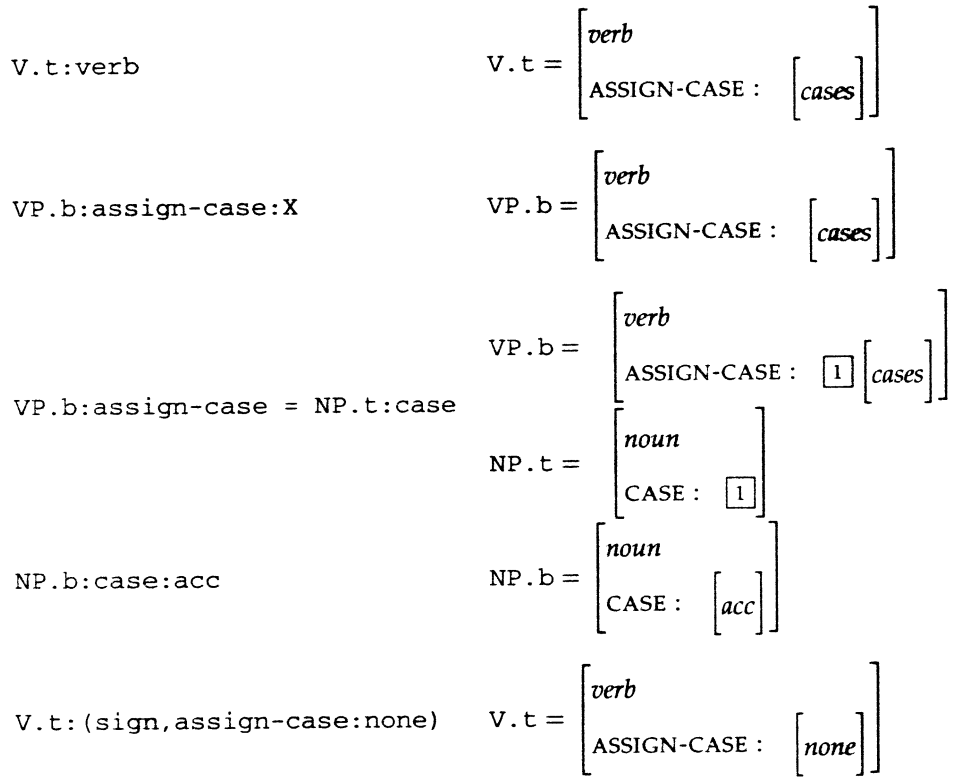


Figure 4
Inferred TFSs.

can be classified into four different types: ambiguous names, typos, undocumented features, and plain errors.

4.2.1 Ambiguous Names. Ambiguous names are an obvious error, but one that is not easy to track without the typing mechanism that we discuss in this article. As the XTAG grammar has been developed by as many as a dozen developers, over 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.[3] 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 in which 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 features 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.

3 Recall that by the feature introduction condition, each feature must be introduced by some most general type (and be appropriate for all its subtypes).

4.2.2 Typos. Another type of error that is very difficult to track otherwise are plain typos. The best example is probably a feature that occurred about 80% of the time as RELPRON and the rest of the time as REL-PRON:
`S_r.t:<relpron> = NP_w.t:<rel-pron>`

4.2.3 Undocumented Features. We have found a great number of features and values that are 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 for better documentation. Of course, the fewer 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.

4.2.4 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 CASEs; and the specification `V.t:<refl> = V_r.b:<refl>` implies that verbs can be REFLexive. Another example is the specification `D_r.b:<punct bal> = Punct_1.t:<punct>`, which handles the balancing of punctuation marks such as parentheses. This should have been either `D_r.b:<punct> = Punct_1.t:<punct>` or `D_r.b:<punct bal> = Punct_1.t:<punct bal>`.

4.3 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 TFS that is output for `PP.b` is depicted in Figure 5 (left). 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 among 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 depicted in Figure 5 (right). 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*.

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)
```

N.t

```
[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]nom,
  definite:[305]bool,
  const:[306]bool,
  rel-clause:[307]bool,
  pron:[308]bool,
  quan:[309]bool,
  gerund:[312]bool,
  refl:[313]bool,
  gen:[314]gens,
  compl:[316]bool)
```

Figure 5
Expanded TFSs.

5. Further Research

We have described in this article a method for validating the consistency of feature structure specifications in grammars that incorporate untyped feature structures. Although the use of feature structures in XTAG is very limited, especially since 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 lexical functional grammars.

We have applied type inference only statically; feature structures that are created at parse time are not validated. By modifying the unification algorithm currently used in XTAG, however, 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 and provide for better representation of feature structures and possibly for more efficient unifications. In a new implementation of XTAG (Sarkar 2000), 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.

Acknowledgments

This work was supported by an IRCS fellowship and NSF grant SBR 8920230. The work of the first author was supported by the Israel Science Foundation (grant number 136/01-1).

References

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

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, England, pages 206–250.

Joshi, Aravind K., L. Levy, and M. Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences* 10(1):136–163.

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, Massachusetts, 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, pages 107–149.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, Chicago, Illinois, and Stanford, California.

Sarkar, Anoop. 2000. "Practical experiments in parsing using tree adjoining grammars." In *Proceedings of the Fifth Workshop on Tree Adjoining Grammars, TAG+ 5*, Paris, France, May 25–27.

Schabes, Yves, Anne Abeillé, and Aravind K. Joshi. 1988. "Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars." In *Proceedings of the 12th International Conference on Computational Linguistics (COLING'88)*, volume 2, pages 579–583, Budapest, Hungary, 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.

XTAG Research Group. 2001. "A lexicalized tree adjoining grammar for English." Technical report IRCS-01-03, Institute for Research in Cognitive Science, University of Pennsylvania, Philadelphia.