Types in Functional Unification Grammars

Michael Elhadad

Department of Computer Science
Columbia University
New York, NY 10027
Elhadad@cs.columbia.edu

December 1989

To appear in Proceedings of the 28th conference of the ACL, 1990.
Abstract: 110 words
Article: 3050 words

Keywords: Unification, FUGs, Types, Generation

Abstract

Functional Unification Grammars (FUGs) are popular for natural language applications because the formalism uses very few primitives and is uniform and expressive. In our work on text generation, we have found that it also has annoying limitations: it is not suited for the expression of simple, yet very common, taxonomic relations and it does not allow the specification of completeness conditions. We have implemented an extension of traditional functional unification. This extension addresses these limitations while preserving the desirable properties of FUGs. It is based on the notions of typed features and typed constituents. We show the advantages of this extension in the context of a grammar used for text generation.

1 Introduction

Unification-based formalisms are increasingly used in linguistic theories [25] and computational linguistics. In particular, one type of unification formalism, functional unification grammar (FUG) is widely used for text generation [14, 17, 2, 20, 19] and is beginning to be used for parsing [15, Kasper87]. FUG enjoys such popularity mainly because it allies expressiveness with a simple economical formalism. It uses very few primitives, has a clean semantics [22, 13, 5], is monotonic, and grants equal status to function and structure in the descriptions.

We have implemented a functional unifier [4] covering all the features described in [14] and [18]. Having used this implementation extensively, we have found all these properties very useful, but we also have met with limitations. The functional unification (FU) formalism is not well suited for the expression of simple, yet very common, taxonomic relations. The traditional way to implement such relations in FUG is verbose, inefficient and unreadable. It is also impossible to express completeness constraints on descriptions.

In this paper, we present several extensions to the FU formalism that address these limitations. These extensions are based on the formal semantics presented in [5]. They have been implemented and tested on several applications.

We first introduce the notion of typed features. It allows the definition of a structure over the primitive symbols used in the grammar. The unifier can take advantage of this structure in a manner similar to [1]. We then introduce the notion of typed constituents and the FSET construct. It allows the declaration of explicit constraints on the set of admissible paths in functional descriptions. Typing the primitive elements of the formalism and the constituents allows a more concise expression of grammars and better checking of the input descriptions. It also provides more readable and better documented grammars.

Most work in computational linguistics using a unification-based formalism (*e.g.*, [24, 27, 12, 14, Kaplan&Bresnan]) does not make use of explicit typing. In [1], Ait-Kaci introduced Ψ-terms, which are very similar to feature structures, and introduced the use of type inheritance in unification. Ψ-terms were intended to be general-purpose programming constructs. We base our extension for typed features on this work but we also add the notion of typed constituents and the ability to express completeness constraints. We also integrate the idea of typing with the particulars of FUGs (notion of constituent, NONE, ANY and CSET constructs) and show the relevance of typing for linguistic applications.

2 Traditional Functional Unification Algorithm

2.1 General idea

The Functional Unifier takes as input two descriptions, called *functional descriptions* or FDs and produces a new FD if unification succeeds and failure otherwise.

An FD describes a set of objects (most often linguistic entities) that satisfy certain properties. It is represented by a set of pairs `[a:v]`, called features, where `a` is an attribute (the name of the property) and `v` is a value, either an atomic symbol or recursively an FD. An attribute `a` is allowed to appear at most once in a given FD F, so that the phrase ''the a of F'' is always non ambiguous [14].

It is possible to define a natural partial order over the set of FDs. An FD X is more specific than the FD Y if X contains at least all the features of Y (that is $X \subseteq Y$). Two FDs are compatible if they are not contradictory on the value of an attribute. Let X and Y be two compatible FDs. The unification of X and Y is by definition the most general FD that is more specific than both X and Y. For example, the unification of `{year:88, time:{hour:5}}` and `{time:{mns:22}, month:10}` is `{year:88, month:10, time:{hour:5, mns:22}}`. When properties are simple (all the values are atomic), unification is therefore very similar to the union of two sets: $X \cup Y$ is the smallest set containing both X and Y. There are two problems that make unification different from set union: first, in general, the union of two FDs is not a consistent FD (it can contain two different values for the same label); second, values of features can be complex FDs. The mechanism of unification is therefore a little more complex than suggested, but the FU mechanism is abstractly best understood as a union operation over FDs (cf [14] for a full description of the algorithm).

Note that contrary to structural unification (SU, as used in Prolog for example), FU is not based on order and length. Therefore, `{a:1, b:2}` and `{b:2, a:1}` are equivalent in FU but not in SU, and `{a:1}` and `{b:2, a:1}` are compatible in FU but not in SU (FDs have no fixed arity) (cf. [16 p.105] for a comparison SU vs. FU).

2.2 Terminology

We introduce here terms that constitute a convenient vocabulary to describe our extensions. In the rest of the paper, we consider the unification of two FDs that we call input and grammar. We define L as a set of labels or attribute names and C as a set of constants, or simple atomic values. A string of labels (that is an element of L^*) is called a path, and is noted $<l_1...l_n>$. A grammar defines a domain of admissible paths, $\Delta \subset L^*$. Δ defines the skeleton of well-formed FDs.

- An *FD* can be an atom (element of C) or a set of features. One of the most attractive characteristics of FU is that non-atomic FDs can be abstractly viewed in two ways: either as a flat list of equations or as a structure equivalent to a directed graph with labeled arcs [11]. The possibility of using a non-structured representation removes the emphasis that has traditionally been placed on structure and constituency in language.

- The meta-FDs NONE and ANY are provided to refer to the status of a feature in a description rather than to its value. `[label:NONE]` indicates that `label` cannot have a ground value in the FD resulting from the unification. `[label:ANY]` indicates that `label` must have a ground value in the resulting FD. Note that NONE is best viewed as imposing constraints on the definition of Δ: an equation $<l_1...l_n>$=NONE means that $<l_1...l_n> \notin \Delta$.
- A *constituent* of a complex FD is a distinguished subset of features. The special label CSET (Constituent Set) is used to identify constituents. The value of CSET is a list of paths leading to all the constituents of the FD. Constituents trigger recursion in the FU algorithm. Note that CSET is part of the formalism, and that its value is not a valid FD. A related construct of the formalism, PATTERN, implements ordering constraints on the strings denoted by the FDs.

Among the many unification-based formalisms, the constructs NONE, ANY, PATTERN, CSET and the notion of constituent are specific to FUGs. A formal semantics of FUGs covering all these special constructs and using simple set theory is presented in [5].

3 Typed features

3.1 A limitation of FUGs: no structure over the set of values

In FU, the set of constants *C* has no structure. It is a flat collection of symbols with no relations between each other. All constraints among symbols must be expressed in the grammar. In linguistics, however, grammars assume a rich structure between properties: some groups of features are mutually exclusive; some features are only defined in the context of other features.

```
                          | Question
                          | Personal
          | Pronoun ------|
          |               | Demonstrative
          |               | Quantified
Noun      |
          | Proper
          |               | Count
          | Common -------|
                          | Mass
```

Figure 1: A system for NPs

Let's consider a fragment of grammar describing noun-phrases (NPs) (cf Figure 1) using the systemic notation given in [28]. The configuration illustrated by this fragment is typical, and occurs very often in grammars.[1] The schema indicates

[1]We have implemented a grammar similar to [28, appendix B] containing 111 systems. In this grammar, more than 40% of the systems are similar to the one described here.

that a noun can be either a pronoun, a proper noun or a common noun. Note that these three features are mutually exclusive. Note also that the choice between the features `{question, personal, demonstrative, quantified}` is relevant only when the feature pronoun is selected. This system therefore forbids combinations of the type `{pronoun, proper}` and `{common, personal}`.

The traditional technique for expressing these constraints in a FUG is to define a label for each non terminal symbol in the system. The resulting grammar is shown in Figure 2.[2] This grammar is, however, incorrect, as it allows combinations of the type `((noun proper) (pronoun question))` or even worse `((noun proper) (pronoun zouzou))`. Because unification is similar to union of features sets, a feature `(pronoun question)` in the input would simply get added to the output. In order to enforce the correct constraints, it is therefore necessary to use the meta-FD NONE (which prevents the addition of unwanted features) as shown in Figure 3.

```
((cat noun)
 (alt (((noun pronoun)
        (pronoun
         ((alt (question personal demonstrative quantified)))))
       ((noun proper))
       ((noun common)
        (common ((alt (count mass))))))))
```

Figure 2: A faulty FUG for the NP system

There are two problems with this corrected FUG implementation. First, both the input FD describing a pronoun and the grammar are redundant and longer than needed. Second, the branches of the alternations in the grammar are interdependent: you need to know in the branch for pronouns that common nouns can be sub-categorized and what the other classes of nouns are. This interdependence prevents any modularity: if a branch is added to an alternation, all other branches need to be modified. It is also an inefficient mechanism as the number of pairs processed during unification is $O(n^d)$ for a taxonomy of depth d with an average of n branches at each level.

3.2 Typed features

The problem thus is that FUGs do not gracefully implement mutual exclusion and hierarchical relations. The system of nouns is a typical taxonomic relation. The deeper the taxonomy, the more problems we have expressing it using traditional FUGs.

We propose extracting hierarchical information from the FUG and expressing it

[2] ALT indicates that the lists that follow are alternative noun types.

```
((alt (((noun pronoun)
        (common NONE)
        (pronoun
         ((alt (question personal demonstrative quantified)))))
       ((noun proper) (pronoun NONE) (common NONE))
       ((noun common)
        (pronoun NONE)
        (common ((alt (count mass))))))))

The input FD describing a personal pronoun is then:
((cat noun)
 (noun pronoun)
 (pronoun personal))
```

Figure 3: A correct FUG for the NP system

```
(define-type noun (pronoun proper common))
(define-type pronoun
  (personal-pronoun question-pronoun
   demonstrative-pronoun quantified-pronoun))
(define-type common (count-noun mass-noun))

The grammar becomes:
((cat noun)
 (alt (((cat pronoun)
        (cat ((alt (question-pronoun personal-pronoun
                    demonstrative-pronoun quantified-pronoun)))))
       ((cat proper))
       ((cat common)
        (cat ((alt (count-noun mass-noun)))))))

And the input: ((cat personal-pronoun))
```

Figure 4: Using typed features

as a constraint over the symbols used. The solution is to define a subsumption relation over the set of constants *C*. One way to define this order is to define types of symbols, as illustrated in Figure 4. This is similar to **Ψ**-terms defined in [1].

Once types and a subsumption relation are defined, the unification algorithm must be modified. The atoms *X* and *Y* can be unified if they are equal OR if one subsumes the other. The result is the most specific of *X* and *Y*. The formal semantics of this extension is detailed in [5].

With this new definition of unification, taking advantage of the structure over constants, the grammar and the input become much smaller and more readable as shown in Figure 4. There is no need to introduce artificial labels. The input FD describing a pronoun is a simple `((cat personal-pronoun))` instead of the redundant chain down the hierarchy `((cat noun) (noun pronoun) (pronoun personal))`. Because values can now share the same label CAT, mutual

exclusion is enforced without adding any pair `[l:NONE]`.[3] Note that it is now possible to have several pairs `[a:v_i]` in an FD F, but that the phrase ‘‘the a of F’’ is still non-ambiguous: it refers to the most specific of the v_i. Finally, the fact that there is a taxonomy is explicitly stated in the type definition section whereas it used to be buried in the code of the FUG. This taxonomy is used to document the grammar and to check the validity of input FDs.

4 Typed constituents: the FSET construct

A natural extension of the notion of typed features is to type constituents: typing a feature restricts its possible values; typing a constituent restricts the possible features it can have.

```
Type declarations:
(define-constituent determiner
  (definite distance demonstrative possessive))

Input FD describing a determiner:
(determiner ((definite yes)
             (distance far)
             (demonstrative no)
             (possessive no)))
```

Figure 5: A typed constituent

Figure 5 illustrates the idea. The `define-constituent` statement allows only the four given features to appear under the constituent `determiner`. This statement declares what the grammar knows about determiners. `Define-constituent` is a completeness constraint as defined in LFGs [Kaplan&Bresnan]; it says what the grammar needs in order to consider a constituent complete. Without this construct, FDs can only express partial information.

Note that expressing such a constraint (a limit on the arity of a constituent) is impossible in the traditional FU formalism. It would be the equivalent of putting a NONE in the attribute field of a pair as in NONE:NONE.

In general, the set of features that are allowed under a certain constituent depends on the value of another feature. Figure 6 illustrates the problem. The fragment of grammar shown defines what inherent roles are defined for different types of processes (it follows the classification provided in [8]). We also want to enforce the constraint that the set of inherent roles is ‘‘closed’’: for an action, the inherent roles are agent, medium and benef *and nothing else*. This constraint cannot be expressed by the standard FUG formalism. A `define-constituent` makes it

[3] In this example, the grammar could be a simple flat alternation ((cat ((alt (noun pronoun personal-pronoun ... common mass-noun count-noun))))), but this expression would hide the structure of the grammar.

```
Without FSET:
(define-constituent inherent-roles
  (agent medium benef carrier attribute processor phenomenon))

((cat clause)
 (alt (((process-type action)
        (inherent-roles ((carrier NONE)
                         (attribute NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process-type attributive)
        (inherent-roles ((agent NONE)
                         (medium NONE)
                         (benef NONE)
                         (processor NONE)
                         (phenomenon NONE))))
       ((process-type mental)
        (inherent-roles ((agent NONE)
                         (medium NONE)
                         (benef NONE)
                         (carrier NONE)
                         (attribute NONE)))))))

With FSET:
((cat clause)
 (alt (((process-type action)
        (inherent-roles ((FSET (agent medium benef)))))
       ((process-type attributive)
        (inherent-roles ((FSET (carrier attribute)))))
       ((process-type mental)
        (inherent-roles ((FSET (processor phenomenon))))))))
```

Figure 6: The FSET Construct

possible, but nonetheless not very efficient: the set of possible features under the constituent `inherent-roles` depends on the value of the feature `process-type`. The first part of Figure 6 shows how the correct constraint can be implemented with `define-constituent` only: we need to exclude all the roles that are not defined for the process-type. Note that the problems are very similar to those encountered on the pronoun system: explosion of NONE branches, interdependent branches, long and inefficient grammar.

To solve this problem, we introduce the construct FSET (feature set). FSET specifies the complete set of legal features at a given level of an FD. FSET adds constraints on the definition of the domain of admissible paths Δ. The syntax is the same as CSET. Note that all the features specified in FSET do not need to appear in an FD: only a subset of those can appear. For example, to define the class of middle verbs (*e.g.*, ‘‘to shine’’ which accepts only a medium as inherent role and no agent), the following statement can be unified with the fragment of grammar given in Figure 6:

```
((verb ((lex "shine")))
 (process-type action)
 (voice-class middle)
 (inherent-roles ((FSET (medium)))))
```

The feature `(FSET (medium))` can be unified with `(FSET (agent medium`

benef)) and the result is (FSET (medium)).

Typing constituents is necessary to implement the theoretical claim of LFG that the number of syntactic functions is limited. It also has practical advantages. The first advantage is good documentation of the grammar. Typing also allows checking the validity of inputs as defined by the type declarations.

The second advantage is that it can be used to define more efficient data-structures to represent FDs. As suggested by the definition of FDs, two types of data-structures can be used to internally represent FDs: a flat list of equations (which is more appropriate for a language like Prolog) and a structured representation (which is more natural for a language like Lisp). When all constituents are typed, it becomes possible to use arrays or hash-tables to store FDs in Lisp, which is much more efficient. We are currently investigating alternative internal representations for FDs (cf. [21, 3, 9] for discussions of data-structures and compilation of FUGs).

5 Conclusion

Functional Descriptions are built from two components: a set *C* of primitives and a set *L* of labels. Traditionally, all structuring of FDs is done using strings of labels. We have shown in this paper that there is much to be gained by delegating some of the structuring to a set of primitives. The set *C* is no longer a flat set of symbols, but is viewed as a richly structured world. The idea of typed-unification is not new [1], but we have integrated it for the first time in the context of FUGs and have shown its linguistic relevance. We have also introduced the FSET construct, not previously used in unification, endowing FUGs with the capacity to represent and reason about complete information in certain situations.

The structure of *C* can be used as a meta-description of the grammar: the type declarations specify what the grammar knows, and are used to check input FDs. It allows the writing of much more concise grammars, which perform more efficiently. It is a great resource for documenting the grammar.

The extended formalism described in this paper is implemented in Common Lisp [4] using the Union-Find algorithm, as suggested in [10, 1, 7] and is used in several research projects [26, 6, 19, 23].

We are investigating other extensions to the FU formalism, and particularly, ways to modify control over grammars: we have developed indexing schemes for more efficient search through the grammar and have extended the formalism to allow the expression of complex constraints (set union and intersection). We are now exploring ways to integrate these later extensions more tightly to the FUG formalism.

Acknowledgments

This work was supported by DARPA under contract #N00039-84-C-0165 and NSF. I would like to thank my advisor Kathy McKeown for her guidance on my work and precious comments on earlier drafts of this paper. Thanks to Tony Weida, Frank Smadja and Jacques Robin for their help in shaping this paper. I also want to thank Bob Kasper for originally suggesting using types in FUGs.

References

[1] Ait-Kaci, H.
A Lattice-theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.
PhD thesis, University of Pennsylvania, 1984.
UMI #8505030.

[2] Appelt, D.E.
Planning English Sentences.
Cambridge University Press, Cambridge, England, 1985.

[3] Boyer, M.
Towards Functional Logic Grammars.
In Dahl, V. and Saint-Dizier P. (editor), *Natural Language Programming and Logic Programming, II*, pages 45-62. North Holland, Amsterdam, 1988.

[4] Elhadad, M.
The FUF Functional Unifier: User's manual.
Technical Report CUCS-408-88, Columbia University, June, 1988.

[5] Elhadad, M.
A Set-theoretic Semantics for Extended FUGs.
Technical Report CUCS-90, Columbia University, 1990.

[6] Elhadad, M., Seligmann, D.D., Feiner, S. and McKeown, K.R.
A Common Intention Description Language for Interactive Multi-media Systems.
In *Presented at the Workshop on Intelligent Interfaces, IJCAI 89*. Detroit, MI, 1989.

[7] Escalada-Imaz, G. and M. Ghallab.
A Practically Efficient and Almost Linear Unification Algorithm .
Artificial Intelligence 36:249-263, 1988.

[8] Halliday, M.A.K.
An Introduction to Functional Grammar.
Edward Arnold, London, 1985.

[9] Hirsh, Susan.
P-PATR: A Compiler for Unification-based Grammars.
In Dahl, V. and Saint-Dizier, P. (editor), *Natural Language Understanding and Logic Programming, II*, pages 63-77. North Holland, Amsterdam, 1988.

[10] Huet, G.
Resolution d'Equations dans des langages d'ordre 1,2,...,ω.
PhD thesis, Universite de Paris VII, France, 1976.

[11] Karttunen, L.
Features and Values.
In *Coling84*, pages 28-33. COLING, Stanford, California, July, 1984.

[12] Karttunen, L.
Radical Lexicalism.
Technical Report CSLI-86-66, CSLI, 1986.

[13] Kasper, R. and W. Rounds.
A Logical Semantics for Feature Structures.
In *Proceedings of the 24th meeting of the ACL*. ACL, Columbia University, New York, NY, June, 1986.

[14] Kay, M.
Functional Grammar.
In *Proceedings of the 5th meeting of the Berkeley Linguistics Society*. Berkeley Linguistics Society, 1979.

[15] Kay, M.
Parsing in Unification grammar.
In Dowty, Karttunen & Zwicky (editor), *Natural Language Parsing*, pages 152-178. Cambridge University Press, Cambridge, England, 1985.

[16] Knight, K.
Unification: a Multidisciplinary Survey.
Computing Surveys 21(1):93-124, March, 1989.

[17] McKeown, K.R.
Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.
Cambridge University Press, Cambridge, England, 1985.

[18] McKeown, K.R. and Paris, C.L.
Functional Unification Grammar Revisited.
In *Proceedings of the ACL conference*, pages 97-103. ACL, July, 1987.

[19] McKeown, K. and M. Elhadad.
A Contrastive Evaluation of Functional Unification Grammar for Surface Language Generators: A Case Study in Choice of Connectives.
In Cecile L. Paris, William R. Swartout and William C. Mann (editors), *Natural Language Generation in Artificial Intelligence and Computational Linguistics*. Kluwer Academic Publishers, 1990.
(also as Columbia technical report CUCS-407-88).

[20] Paris, C.L.
The Use of Explicit User models in Text Generation: Tailoring to a User's level of expertise.
PhD thesis, Columbia University, 1987.

[21] Pereira, F.
A Structure Sharing Formalism for Unification-based Formalisms.
In *Proceedings of the 23rd annual meeting of the ACL*, pages 137-144. ACL, 1985.

[22] Pereira, F. and S. Shieber.
The Semantics of Grammar Formalisms Seen as Computer Languages.
In *Proceedings of the Tenth International Conference on Computational Linguistics*, pages 123-129. ACL, Stanford University, Stanford, Ca, July, 1984.

[23] Robin, J.
Lexical Choice in COMET.
Technical Report, Columbia University, 1990.

[24] Sag, I.A. and Pollard, C.
Head-driven phrase structure grammar: an informal synopsis.
Technical Report CSLI-87-79, Center for the Study of Language and Information, 1987.

[25] Shieber, S.
CSLI Lecture Notes. Volume 4: *An introduction to Unification-Based Approaches to Grammar*.
University of Chicago Press, Chicago, Il, 1986.

[26] Smadja, F.A. and McKeown, K.R.
Automatically Extracting and Representing Collocations for Language Generation.
Technical Report, Columbia University, 1990.
Submitted to ACL90.

[27] Uszkoreit, H.
Categorial Unification Grammars.
Technical Report CSLI-86-68, Center for the Study of Language and Information - Stanford University, 1986.

[28] Winograd, T.
Language as a Cognitive Process.
Addison-Wesley, Reading, Ma., 1983.