CUCS-420-89

Extended Functional Unification ProGrammars

Michael Elhadad
Columbia University, Dept. of Computer Science
450 Computer Science Bdg.
New York, NY 10027
elhadad@cs.columbia.edu

Abstract: 190 words
Article: 4700 words

Area of Submission: C1, natural language.
Keywords: Natural Language Processing, Generation, Unification, FUGs

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 adapted to the expression of simple yet very common taxonomic relations and it does not allow easy manipulation of complex data-structures like lists or sets. We present in this paper a set of extensions that keep the desirable properties of the formalism but make it more flexible and easier to use. We first introduce the notion of typed features and typed constituents. Types define a structure over the set of primitive symbols used by the formalism. We then introduce extended unification: specialized unification methods can be defined for user-defined data-types. This extends the power of the system to handle complex data-structures efficiently. Taking advantage of a structured set of primitives and of specialized unification methods, the resulting formalism is more flexible, easier to use and produces better documented grammars than traditional functional unification. It can therefore be used to address deeper levels of text generation than was possible before.

1 Introduction

Unification-based formalisms are increasingly used in linguistic theories [16] and for natural language applications. In particular, functional unification grammars (FUGs) are widely used for text generation [10, 12, 1, 13, 14] and are starting to be used for parsing [11, 8]. FUGs enjoy such popularity mainly because they ally expressiveness with a simple economical formalism. It uses very few primitives, has a clean semantics [15, 9], is monotonic, and grants equal status to function and structure in the descriptions.

Having worked with the functional unification (FU) formalism, we have found all these properties very useful; but we also have met with limitations. The FU formalism is not adapted to the expression of simple yet very common taxonomic relations. The traditional way to implement such relations in FUG is verbose, inefficient and not readable. We also have applied FUGs to non-traditional applications, beyond surface generation and syntactic parsing. The task we have investigated is the choice of connectives between two propositions - a problem at the junction of deep and surface generation [4, 5, 13]. For this problem, we had to express constraints between sets, check set membership, and compute set unions and intersections. For this task, and others involving complex data structures, the basic FU formalism is awkward, and sometimes impossible, to use.

In this paper, we present a set of extensions to the FU formalism that we have defined and implemented. Our goal is to retain the nice properties of the formalism but make it more flexible and versatile.

We first introduce the notion of typed features and of typed constituents. The idea is to define a structure over the primitive symbols used in the grammar and to take advantage of this structure when unifying terms. The next step is to define specialized unification methods for user-defined data types. Specialized unification methods actually define procedurally a structure over non-atomic expressions. We propose a way to smoothly integrate such methods in the overall FU formalism. In the extended formalism, a grammar has two parts: a type definition and a functional description. We call such a combination a ProGrammar.

Typing the primitive elements of the formalism allows a more concise expression of grammars, allows better checking of the input descriptions, and provides more readable and better documented grammars. User-defined unification methods permit the unifier to efficiently handle complex data-structures like lists or sets, and to perform computations that are beyond the power of standard unification methods, like arithmetic operations. In general, our idea is to remove the burden of expressing complex constraints from the formalism and put it on an environment better suited for their expression. The problem our extension addresses is how to properly integrate such foreign environments within the FU formalism.

In the rest of the paper, we first describe the traditional functional unification algorithm. We then introduce the notion of typed features and finally describe an example of user-defined unification methods.

2 Traditional Functional Unification Algorithm

2.1 General idea

The Functional Unifier (FU) 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 [10].

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 [10] for a full description of the algorithm). In Appendix I, we give a detailed description of the unification 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.

2.2 Terminology

We introduce here terms that will constitute a convenient vocabulary to describe the algorithm and its 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 $<l_1 \ldots l_n>$.

An **FD (functional description)** 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 [7]. The possibility to use a non-structured representation removes the emphasis that has traditionally been put on structure and constituency in language.

The **total-FD** is the FD that will eventually result from the unification. It is the reference for all paths and contains all known information during unification.

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 real value in the total-FD. [label:Any] indicates that **label** must have a real value in the total-FD. A real value is either an element of *C* (a constant) or a complex FD containing at least an element of *C* at some level. Non real values are meta-FDs and NIL or complex FDs containing only NIL or meta-FDs.

Alternation: FDs seen so far contain an implicit conjunction; **{size:small, color:red}** describes objects that are small and red. An alternation is a disjunctions of features. Each disjunct is called a branch of the alternation. For example, **{size:small, color:{alt:{red, blue, white}}}** specifies that the color can be either red, blue or white.

A **constituent** of a complex FD is a distinguished subset of features. In linguistic descriptions, features describing a sentence as a whole need to be distinguished from those describing a constituent of the sentence. A special label called Cset (Constituent Set) is used for this purpose. The value of the label Cset is a list of paths leading to all the constituents of the FD. Constituents trigger recursion in the FU algorithm as described in the appendix. Note that Cset is part of the formalism, and that its value is not a valid FD.

A **grammar** is an FD, containing disjunctions and meta-fds. A grammar describes the set of FDs compatible with it. Traditionally, a grammar is a big alternation containing a branch for each grammatical category. The category of an FD is specified by the value of the special label **cat**. **Grammar** = **{alt:{{cat:c1, ...}, {cat:c2, ...}, ...}}**.

3 Typed features and constituents

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 relation 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.

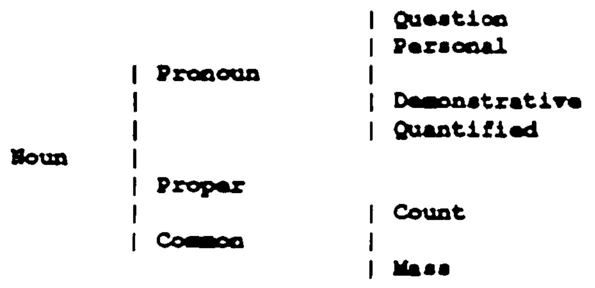

Figure 1: A system for NPs

Let's consider a fragment of grammar describing noun-phrases (NPs) (cf figure 1). We use here the systemic notation given in [17]. The configuration illustrated by this fragment is typical, and occurs very often in grammars.[1] The schema indicates 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}.

```
((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

```
((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

The traditional technique to express 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. This grammar is, however, incorrect, as it allows combinations of the type ((noun proper) (pronoun question)) or even worse ((noun proper) (pronoun zouzou)). In order to enforce the correct constraints, it is necessary to use the meta-fd None as shown in figure 3.

There are now two problems with this corrected FUG implementation. First, both the input FD describing a pronoun and the grammar are redundant and longer than

[1]We have implemented a grammar similar to [17, appendix B] containing 94 systems. In this grammar, more than 40% of the systems were similar to the one described here.

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 are the other classes of nouns. 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 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.

```
(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

We propose extracting hierarchical information from the FUG and expressing it 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.

Once types and a subsumption relation are defined, the algorithm needs to be adapted (the **atom-unify** function in the appendix). 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.

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 now 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].[2] 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 can be used to document the grammar and to check the validity of input FDs.

3.3 Typed constituents

A natural extension of the notion of typed features is to type constituents: typing features restricts the possible values of the feature; typing constituents restricts the possible features of a constituent. Typing declares that only certain features are relevant for a given constituent.

Type declarations:

```
(define-constituent determiner
  (definite distance demonstrative possessive))
(define-feature definite (yes no))
(define-feature distance (far near))
(define-feature demonstrative (yes no))
(define-feature possessive (yes no))
```

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**. Similarly, the **define-feature** statements specify the allowed values for the given features. These statements say what the grammar knows about determiners. **Define-constituent** is a completeness constraint as defined in LFGs [6]; it tells what the grammar needs in order to consider a constituent complete.

Note that expressing such a constraint 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**.

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 to provide good documentation of the grammar. Typing also

[2]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.

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: the 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. The manipulation of FDs is then much more efficient. We are currently investigating alternative internal representations for FDs.

4 Extended unification

4.1 An example: list handling

List or set values are often useful in grammatical descriptions. For example, to describe the use of conjunctions like ''and'' or ''or,'' it is convenient to manipulate lists. A conjunction creates a complex syntactic group out of a list of constituents of the same category. For example, from the noun groups ''Bill and Joe'' and ''Mary and Janet,'' one can create the single group ''Bill, Joe, Mary and Janet.'' Similarly if a list of verb groups all share the same tense and mood, it is possible to conjoin them. One way of modeling this phenomenon is to represent the original phrases in lists and compute the resulting complex phrase as the ''append'' of these lists.

4.2 A limitation of FUGs: expression of complex constraints

FUGs can currently be used to express this kind of complex constraint, but we argue that they should not be used for that. The FU formalism defines complex constraints by composing simple constraints and conflations[3] using conjunction and disjunction (negation is also sometimes allowed [7]).

Figure 6 shows this scheme applied in a grammar implementing **append** that behaves exactly like its Prolog counterpart (it is bidirectional). This grammar works but it is not very readable. We give here just a few indications of how it works. Lists are represented as FDs with the constituents car and cdr in a classical manner, but with the FD None playing a role equivalent to the Lisp atom NIL. The notions of environment and variable in Prolog correspond to the notion of total-FD and path in FU. The total-FD contains the environment of a computation. Variables are then just places or positions within the total-FD, and are referred to using paths. Finally, constituents in FU play the role of arguments to procedures. For example, the procedure **append** has three arguments, X, Y and

[3]A conflation is a feature of the form <l1...ln>=<k1...km>. It forces the value of the two paths to be unified.

```
((cat append)
 (alt
     ;; 1st branch: append([],Y,Y).
     (((x None) (z (^ y)))
      ;; 2nd branch: append([X/Xs],Y,[X/Z]):-append(Xs,Y,Z).
      ;; recursive call to append with new arguments x, y.
      ((cset (z))
       (z ((car (^ ^ x car))
           (cdr ((cat append)
                 (x (^ ^ ^ x cdr))
                 (y (^ ^ ^ y))))))))))
 ;; Normalize: append must contain CAR and CDR of the result.
 (car (^ z car))
 (cdr (^ z cdr)))[4]

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Figure 6: A grammar for append and the equivalent Prolog program

Z.

The main problem with this implementation, not to mention its readability, is that there is no notion of "environment" besides the total-FD. Therefore, when a program works recursively, all the local variables that are normally stacked in an external environment are stacked within the total-FD. At the end, the total-FD contains the whole stack of the computation, and is pretty awkward to manipulate. Even when we are just interested in computing a result, the FUG implementation will return the whole computation leading to the result.

4.3 Extended unification

When introducing typed features, we have defined a structure over the set of constants *C*, but we have also modified the unification algorithm (the **atom-unify** function). In other words, we have defined a new unification method, distinct from the classical default method, and specialized in handling atoms. There are actually other specialized methods built into the traditional algorithm: Csets and patterns, which express ordering constraints between the constituents and have a special syntax, are handled by distinct specialized procedures, as their values are not "legal" FDs. It is a natural idea to extend this behavior to user-defined data-types. We explain now how to allow the definition of new unification methods specialized in handling user-defined data-types.

Figure 7 illustrates the approach we have taken to integrate non-FD items within the FU algorithm. In exactly the same way as patterns are unified by a special function, we can define new functions (written in the underlying language, in our

[4]The notation ^ is used for relative paths. Relative paths are best understood when considering the graph representation of FDs. To find the reference of a relative path in a graph, start at the position where the path appears, then go one arc up for each ^ and then follow down the arcs as indicated by the labels. Relative paths are important to make grammars "relocatable."

Type declarations:
```
(define-type list :method list-unify :member is-list)
(defun list-unify (l1 l2 fail success) <body of function>)
(defun is-list (l) <body of predicate>)

(define-type append :method append-unify :member is-append
                    :specialize list)
(defun append-unify (l1 l2 fail success) <body of function>)
(defun is-append (expr) <body of predicate>)

(define-feature a list)
(define-feature b list)
(define-feature c append)
```
Using the types list and append:
```
(unify '((a [?x ?y ?x ?y]) (b [1 ?z 1 2])) '((a <b>))
       --> ((a [1 2 1 2]) (b <a>))
(unify '((a [1 2]) (b [3 4])) '((c (<a> <b>))))
       --> ((a [1 2]) (b [3 4]) (c [1 2 3 4]))
```
Figure 7: Specialized method for positional unification and append

case Lisp) that perform a different unification method. We use the example of positional unification on lists (traditional unification where ?x stands for a variable). A type is defined by two functions: a membership predicate checks the syntax of an expression and decides whether it is a member of the type or not; a unification method implements a subsumption relation between the elements of the type. For example, the type append in figure 7, is defined by the predicate is-append. It has roughly the same syntax as Csets: it accepts lists of paths, that all point to elements of type list, or directly a list. The type append is also defined as a subtype of the type list by the :specialize statement.

To compute the append of two lists appearing at level <a> and <b>, we add a feature [c=(<a> <b>)] in the grammar. Note that this append is also bidirectional. The constraint expressed by the patterns in figure 7 would be very difficult to express with only the standard FU notation. The types list and append therefore significantly increase the flexibility of the unifier and make the formalism much more versatile.

Note that we consider user-defined types only in leaf positions within the total-FD, or in other words, the sub-components of a user-defined type are not accessible from the outside. Conceptually, it means that all the types we define are subsets of the set of ''constants'' *C*, and that, as far as the unifier is concerned, the only way to build a structure is to use a legal FD. The subsumption relation within the new defined types is defined procedurally by the unification methods. This extension is therefore very much in the same spirit as typed features, although it provides great flexibility from a computational perspective.

If the unification method is non deterministic, the handling of backtracking must be well integrated with the rest of the unifier. We use the scheme of success continuations as presented in [2] for the handling of backtracking. This is a simple method, that only requires the unification method to use two extra arguments (fail and success) which are continuations. We have written a library of Lisp functions that facilitate the writing of such functions. The protocol of communication is therefore well designed, and we have found it easy to write a variety of unification methods.

5 Conclusion and future work

Functional Descriptions are built from two components: a set *C* of primitives and a set *L* of labels. All the structuring of FDs is done by using strings of labels. We have showed in this paper that there is much to be gained by relegating some of the structuring power to the set of primitives. The set *C* is no longer considered a flat set of symbols, but is viewed as a richly structured world, containing even non-atomic individuals. The structure we use is a subsumption relation, that can be defined explicitly, using discrete types of atoms, or procedurally, using a specialized unification method. In our extended formalism, a grammar has two components: a type definition over *C* and an FD. We call this combination a ProGrammar.

The structure of *C* is a great resource for documenting the grammar. It 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 writilg of much more concise grammars, which perform more efficiently. It allows the expression of complex constraints in a flexible way.

The extended formalism described in this paper is implemented in Common Lisp [3]. The added flexibility of ProGrammars has allowed us to address aspects of text generation (deep generation) that were beyond the power of standard FUGs. Using the extended formalism, we have implemented a procedure of connective selection [5] - a problem at the junction of deep and surface generation. Our method involves constraints such as set intersection (to determine whether the propositions to be connected have a topic in common), and checking that propositions are argumentatively consistent. The ProGrammar implementation of this procedure is efficient and has been easy to design.

We are currently investigating other extensions to the FU formalism, and particularly, ways to modify the control over grammars: we are working on indexing schemes that will allow a more efficient search through the grammar.

Acknowledgments

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

I. The Unify algorithm

The main function of the unification algorithm is **unify(input, grammar)**. We give here a detailed description of important auxiliary functions. **Graph-unify** is the most complex function. It unifies two FDs at the top-level. Basically, **graph-unify** enriches the input FD with all attribute-value pairs of the grammar that are not already in the input and recursively unifies the values of pairs that exist in both the input and the grammar. **Graph-unify** traverses the whole grammar depth-first, and calls the appropriate specialized function when it reaches the leaves: **Atom-unify** is called when one of the FDs is an atom. It succeeds if the two FDs are equal; it also implements the semantics of the meta-fds; **Pattern-unify** is a specialized function to handle patterns, which are special ordering constraints and have a form similar to Csets; **Cset-unify** handle Csets. Most of the complexity of **Graph-unify** is due to the handling of conflations.

Unify calls first **graph-unify** on its arguments, performing a first sweep of unification through the input. Then, it identifies the constituents of the result of the first sweep (the Cset), and recursively unifies each of them. All the constituents of the total-FD, at all levels are therefore traversed in a breadth-first manner and unified with the grammar, each according to its grammatical category. It is therefore possible to describe recursive structures in the FU formalism. Note that **Graph-unify** is non-deterministic.

```
Unify(input, grammar)
    input   : an FD with no disjunctions and no meta-fds.
    grammar : an FD.
Let   total-FD be graph-unify(input, grammar, <>)
    cset    be the constituent set of total-FD
for each of the constituents in cset do
    let fd   = graph-unify(constituent, grammar, path)
    let cset = append(cset, cset[fd])

Graph-unify(fd1, fd2, path)
    fd1  : sub-fd of input
    fd2  : an fd with possible disjunctions and meta-fds
    path : level of fd1 in total-FD
If atom(fd2) return atom-unify(fd1, fd2, path)
else for each pair of fd2, label:value2 DO
    if (label = ALT) alt-unify(total-FD, value2, path)
    else
    if there is no pair in fd1 enrich(total-FD, label:value2, path)
    else there is a pair label:value1 in fd1
       if (label = PATTERN)
            let value be Pattern-unify(value1, value2)
              enrich(total-FD, label:value, path)
         (label = CSET)
            let value be Cset-unify(value1, value2)
              enrich(total-FD, Cset:value, path)
         (value1 is a path and value2 is a path)
            let p1 be the fd pointed to by value1 in total-FD
              p2 be the fd pointed to by value2 in total-FD
              value be Graph-unify(p1, p2, value1)
            enrich(total-FD, value1=value)
            enrich(total-FD, path=value1)
            enrich(total-FD, value2=value1)
         (value1 is a path and value2 is not a path)
            let p1 be the FD pointed to by value1 in total-FD
              value be Graph-unify(p1, value2, value1)
            enrich(total-FD, value1=value)
            enrich(total-FD, path=value1)
          (value1 is not a path and value2 is a path)
            let p2 be the fd pointed to by value2 in total-FD
              value be Graph-unify(p2 , value1, value2)
            enrich(total-FD, value2=value)
            enrich(total-FD, path=value2)
          (normal case: value1 and value2 are valid fds)
            let value be Graph-unify(value1, value2, path|label)
            enrich(total-FD, label:value, path)

Atom-unify(fd1, fd2, path): fd
    fd1 : arbitrary sub-fd of input
    fd2 : atom - element of A or NIL or a meta-fd
    path: level of fd1 in the total-FD
if (fd1 = fd2)  return fd1
  (fd2 = NIL)  return fd1
  (fd2 = Any)  if fd1 is a real-value return fd1 else mark Any-P[path] and return Any
  (fd2 = None)  if fd1 is a real-value fail ELSE return None
  (fd2 = Given) if value-of-path(input, path) is a real-value return fd1 else fail
else fail
```

References

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

[2] Carlsson, M.
On Implementing Prolog in Functional Programming.
In *Symposium on Logic Programming*, pages 154-159. IEEE, 1984.

[3] Elhadad, M.
The FUF Functional Unifier: User's manual.
Technical Report, Columbia University, June, 1988.

[4] Elhadad, M. and McKeown, K.R.
What do you need to produce a 'but'.
Technical Report CUCS-334-88, Columbia University, January, 1988.

[5] Elhadad, M. and McKeown, K.R.
A Procedure for the Selection of Connectives in Text Generation.
1989.
Submitted to ACL conference 1989.

[6] Kaplan, R.M. and J. Bresnan.
Lexical-functional grammar: A formal system for grammatical representation.
The Mental Representation of Grammatical Relations.
MIT Press, Cambridge, MA, 1982.

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

[8] Kasper, R.
Systemic Grammar and Functional Unification Grammar.
Systemic Functional Perspectives on discourse: selected papers from the 12th International Systemic Workshop.
Ablex, Norwood, NJ, 1987.

[9] 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.

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

[11] Kay, M.
Parsing in Unification grammar.
Natural Language Parsing.
Cambridge University Press, Cambridge, England, 1985, pages 152-178.

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

[13] McKeown, K. and M. Elhadad.
Comparison of Surface Language Generators: a Case Study in Choice of Connectives.
Proceedings of the 4th Workshop on Language Generation.
Forthcoming, 1988.
Forthcoming.

[14] 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.

[15] 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.

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

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