A Prolog implementation of the Functional Unification Grammar Formalism

Massimo Fasciano
Guy Lapalme
Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, Succ. A
Montréal, Québec, Canada, H3C 3J7
e-mail: {fasciano,lapalme}@iro.umontreal.ca

March 29, 1994

Abstract

This paper compares the use of Lisp and Prolog for the implementation of a functional grammar unification system. To achieve this comparison, we have taken as a starting point Michael Elhadad's FUF system, which is written in Lisp and produced a much smaller and more efficient Prolog version (PFUF) retaining many of FUF's essential features. Our approach is based on a precompilation scheme that reduces most of the runtime overhead.

1 Introduction

Since its introduction in the early 80's [Kay79, Kay85], the functional unification grammar formalism has rapidly gained acceptance in the field of text generation [M+90].

Functional unification grammars display the following characteristics, which separate them from the standard context-free models such as the Prolog DCG formalism (see [GM89] for details):

- The use of features (gender, number, etc...) to constrain rule selection.
- The use of flexible constraints (patterns) to specify the order of the terminals in the linearized form (natural language sentence).
- Automatic unification of sub-constituents (no explicit recursive calls in the grammar itself).

Structural grammars such as the Prolog DCG use fixed-arity terms to represent functional structures (sets of features). Each feature is identified by its position within a term. This approach has two major drawbacks: first, all features must be specified because the terms have a fixed arity, thus producing large structures with "holes" representing missing features; second, adding or removing a single feature in a rule requires updating many more rules because this changes the arity of the term that holds the feature set.

On the other hand, functional grammars name their features, so that they can be referred to by name instead of by position. Fixed-arity terms are no longer required.

Specifying the order of constituents in a DCG-style grammar also brings its share of problems. Indeed, constituent order is specified in too rigid a manner, which often forces the user to explicitly list all possible orderings.

Functional grammars solve this problem by introducing patterns. A pattern is used to specify constituent order in a less restrictive manner. From a set of partial ordering constraints imposed by the patterns, the system automatically calculates a possible linear representation (NL sentence) of the functional structure.

The third interesting feature of some functional unification grammar formalisms such as FUF is the use of constituent sets (CSETs) to specify recursive unifications.

Structural grammars explicitly call the rules to use for recursive unification. The order of these calls is also used to specify constituent order in the linear structure.

Functional grammars usually have no use for explicit recursive calls, since the order of the constituents is determined via a pattern. In this case an automated approach is more desirable. Most of the time, the nature of a constituent indicates if it requires recursive unification. The set of recursively-unified constituents can also be specified explicitly in the grammar.

This has the advantage of being more declarative than explicit recursive rules, thus allowing the implementation to use different control strategies. Indeed, once it is known that a constituent has to be unified, the system can choose when to do it.

2 The FUF system

The FUF system [Elh91] was designed by Michael Elhadad at Columbia University. It is written entirely in Common Lisp, which makes it very portable.

It uses functional structures (feature sets) to represent the grammar and the user input. In fact, the whole grammar is a single feature set. There are no rules per se, which makes it very different from most systems based on Prolog. All of the features of conventional FUGs (CSETs, patterns, disjunctions, the special value "any") are implemented, plus a number of extensions (special value "given").

The management of flexible order constraints is implemented through the use of patterns that allow "holes" of varying size. See sections 3 and 4 for details.

The basic inference engine of FUF controls the unification process based on this simple algorithm:

1. The functional structure given as input is unified with the whole grammar. This process "enriches" the input.

2. Each constituent in the enriched input is then recursively unified with the whole grammar if it is a member of the constituent set (CSET).

The CSET is determined as follows:

- If the meta-feature `cset` is present in the top-level structure, then the list of constituents that it specifies becomes the CSET.

- Otherwise, the CSET is the union of

  - all the sub-constituents which contain the `cat` feature (category)
  - all the sub-constituents mentioned in the `pattern` meta-feature

The default option for CSET determination is almost always sufficient.

3 The PFUF prototype

Our prototype (PFUF) was implemented in Prolog because we felt that it was a better implementation language for a unification system. The result (cf. section 5) is a smaller, more efficient system.

PFUF is based on FUF, so it tries to stick as close to FUF's formalism as is possible within a Prolog framework. Indeed, it also uses functional structures, order patterns and automatic recursive unifications. Nevertheless, some aspects of FUF have been changed to take into the account the logic-based nature of our system. For example, as can be seen in section 4, our system doesn't put the whole grammar in a single monolithic structure. It uses Prolog rules to make it more modular.

PFUF's functional structure unifier is based on [Boy87] with a number of extensions, the most important being the use of alternatives (disjunctions) within the grammar. An interesting discussion of the basic model can be found in [GM89].

Our pattern unifier closely resembles FUF's. Indeed, patterns are allowed to contain holes of unspecified size. For example, to specify that the subject starts the sentence and is followed somewhere in the sentence by a verb, you would use the following pattern: `[subject,...,verb,...]`. Later on, to add an object right after the verb, you just add another pattern (without changing the first one): `[...,verb,object,...]`. To add an indirect object at the end of the sentence, simply add `[...,iobject]`. The resulting pattern after automatic linearization would look something like `[subject,...,verb,object,...,iobject]`.

The inference engine at the heart of our system tries to reproduce FUF's behavior as closely as possible. Nevertheless, it is much simpler. This is due to the format of our rules. Indeed, a PFUF grammar is a set of directly executable Prolog rules, capable of doing their own unifications. Thus, the inference engine's role is very limited: it calls the rule associated with the top level grammar, giving it the user's request as input. The grammar then enriches (by unification) the input, and then control is transferred back to the engine (procedure call returns). The engine's only real time-consuming job is to determine the constituents that require recursive unification and to call the grammar again with each of them as input. The advantages of this approach are that the inference engine is small and easily customizable and that most of the work is done by very efficient precompiled Prolog code in the rules, instead of requiring a meta-interpreter.

The simplicity and efficiency of the runtime layer is due mainly to the precompiler (see section 4.2).

4 Example grammar

In this section, we give an overview of syntactic differences between FUF and our prototype PFUF. These differences are shown through excerpts of a small grammar (`gr3.l`) from the FUF 5.1 distribution, which we translated to PFUF syntax. Numbers between square brackets have been inserted at key points in the code for easy reference.

4.1 FUF

FUF 5.0 grammars are written as a Lisp list, which is assigned to the global variable `*u-grammar*` (cf. `[1]`). The format used to represent *feature/value* pairs is very common in Lisp systems: a pair is given as a 2-element list (ex: `(cat clause)`) and pairs are combined using standard Lisp lists to produce functional structures.

In our opinion, the monolithic nature of FUF 5.0[1] grammars (one big functional structure), makes them very hard to read.

The grammar starts with an alternative between the top-level categories (clause, verb, etc...). Alternatives are represented via the `alt` meta-feature in a structure (cf. `[2]`). The use of a feature to represent a disjunction is a little strange, but given the monolithic nature of the grammar, it is unavoidable. It does make the grammar even harder to read by introducing another level of parentheses.

The reserved value `none` is used to specify that a feature has no value (cf. `[3]`). The reserved value `given` indicates that the feature must have been specified in the input (cf. `[4]`).

[1] FUF 5.1 introduces a way of defining grammars in a modular way with `def-alt` and `def-conj`. Named disjunctions and conjunctions can be defined and are used in a grammar by using the syntax `(:! name)` and `(:& name)` respectively.

```
(defun gr3 ()
  (setq *u-grammar*   ; [1]
'((alt                ; [2]
   (
    ((cat clause)
     (process-type actions)
     (prot ((alt (none ((cat np) (animate yes))))))
     (goal ((alt (none ((cat np))))))   ; [3]
     (benef ((alt (none ((cat np))))))
     (verb ((process-class actions)
            (lex given)))               ; [4]
     (alt voice (:index voice)          ; [5]
         ;; Voice active
         (((voice active)
           (verb ((voice active)))
           (subject {^ prot})
           (object {^ goal})            ; [6]
           (iobject {^ benef}))
          ;; Voice passive
          ((voice passive)
           (verb ((voice passive)))
           (alt
              ;; Is there an explicit prot
              ;; in the input?
               (((prot none)
                 (by-obj none))
                ((prot given)
                 (by-obj ((np {^ ^ prot})))))  ; [7]

                    ⋮

     ;; Arrange order of complements
     (pattern (subject verb dots object dots))  ; [8]
     (alt verb-voice (:index (verb voice))
       ;; VERB VOICE ACTIVE
       (((verb ((voice active)))
         (alt dative
             ;; John gave Mary the book
             (((verb ((transitive-class bitransitive)
                      (dative-prep none)))
               (pattern (dots verb iobject object dots)))  ; [9]
              ;; John gives a book to Mary
              ((verb ((dative-prep given)))
               (dative ((cat pp)
                        (prep ((lex {^ ^ ^ verb dative-prep})))
                        (np {^ ^ iobject})))
               (pattern (dots verb object dative dots)))   ; [10]
              ;; Catch all for non-bitransitive cases
              ((verb ((dative-prep none)))))))

                    ⋮
```

Figure 1: FUF 5.0 version of the grammar

```
gram(CL) :-
    given(CL:verb:lex),                                  % [1]
    CL === [cat:clause,process_type:actions,
            verb:[process_class:actions]],
    CL:prot === (none ; [cat:np,animate:yes]),           % [2]
    CL:goal === (none ; [cat:np]),
    CL:benef === (none ; [cat:np]),
    clause_voice(CL),
    transitive_class(CL),
    CL:verb:cat === verb_group,
    CL:subject:number === CL:verb:number,                % [3]
    clause_pattern(CL).

clause_pattern(CL) :-
    CL:pattern === [subject,verb,...,object,...],        % [4]
    CL:verb:voice === Voice,
    clause_pattern(Voice,CL).                            % [5]

clause_pattern(active,CL) :-
    % bitransitive verb without preposition
    CL === [verb:[transitive_class:bitransitive,
                  dative_prep:none],
            pattern:[...,verb,iobject,object,...]];      % [6]
    % preposition given in input
    given(CL:verb:dative_prep),
    CL === [dative:[cat:pp],
            pattern:[...,verb,object,dative,...]],       % [7]
    CL:dative:prep:lex === CL:verb:dative_prep,
    CL:dative:np === CL:iobject;
    % default case
    CL === [verb:[dative_prep:none]].

clause_pattern(passive,CL) :-

                ⋮
```

Figure 2: PFUF version of the grammar

Alternatives can be named and one can ask the system to index on a given feature (for efficiency reasons) (cf. `[5]`). One usually asks for such an indexation on a feature which is highly discriminating for the selection of the correct alternative.

To unify 2 features equationnally, one can use the following syntax: `(feature1 {^ feature2})` (cf. `[6]`). This indicates that the features `feature1` and `feature2` of the current structure must have the same value (or be made to by unification). By adding an appropriate number of `^`, one can specify unification with a feature situated above the current one in the top-level structure. For example, `(np {^ ^ prot})` unifies the current level's `np` feature with the parent level's `prot` feature (cf. `[7]`). A more rarely-used equational notation without the `^` can also be used to specify absolute unifications (from the top of the structure) instead of relative ones.

A little later in the grammar, we find the constraints on constituent order. The `pattern` meta-feature is used to specify them, with the reserved value `dots` indicating a variable-sized sequence of constituents (cf. `[8]`, `[9]`, `[10]`).

4.2 PFUF

Our system represents a grammar as a set of Prolog rules. The entry point of the grammar is the Prolog predicate `gram/1` whose single parameter is the input functional structure.

The syntactic representation we chose for *feature/value* pairs is a very common one in the Prolog world: each pair is represented as *feature* : *value* and the pairs are combined in a standard Prolog list to produce a functional structure. Unifications are performed equationnally using the "`===`" operator (cf. `[3]`).

The predicate `given` is used to test if a feature is defined in the input structure (cf. `[1]`). It replaces FUF's reserved value of the same name. It's implementation is very straightforward in Prolog and requires no modifications of the basic unifier.

Disjunctions are usable at 2 levels. At the rule level, the standard Prolog disjunction operator "`;`" can be used directly, since rules are Prolog code (cf. `[6]`). At the level of the unifier, the "`===`" operator allows the use of "`;`" to specify a disjunction of possible values in a more compact way (cf. `[2]`). Note that this second form of disjunction is translated by our rule pre-processor into the first form.

Contrary to FUF 5.0, our grammars are very modular because they are separated into many small Prolog rules that are specialized for a given task (ex: `clause_pattern`). This makes the grammar much easier to read and maintain. Also, since our rules are represented directly as Prolog code, it is very easy to call native Prolog procedures from the grammar. No "escape" mechanism is necessary.

Indexation of alternatives is provided by the Prolog compiler, and we try to precompile

the grammar into a form that makes it possible for almost any indexing compiler to optimize it (see the transformation shown in figure 4).

Constituent order specification is done as in FUF with the meta-feature `pattern` (cf. `[4]`, `[6]`, `[7]`). The atom "..." represents an unspecified number of constituents (cf. section 3).

To make the runtime layer more efficient, our system preprocesses the input grammar and transforms it into Prolog code that can run almost directly (CSET calculations are done by an external inference engine). Many tranformations are applied, including the following:

- transform feature-level disjunction into rule-level disjunction.
  `(ex: X===(A;B) ---> X===A;X===B)`
- transform calls to `===` into direct calls to the feature unifier `unif`.
- partial evaluation of functional structures to reduce the number of calls to `unif`.
  `(ex: (X === [a:1,b:2], X === [c:3,d:4]) ---> (X === [a:1,b:2,c:3,d:4]))`
- generate new disjunctive rules to replace inter-rule disjunction (figures 3 and 4).
- partial evaluation of simple constraints to bind the new rules' parameters, thus allowing Prolog to index efficiently (figure 4).
- translate calls to `given` into `unif/nonvar` pairs (figure 3).

Figure 5 shows a typical input that can be fed to our system. This input, which describes the sentence "Mary is thrown a heavy ball by John", was translated directly from example `ir33` in the FUF 5.1 distribution. It is written in structural notation.

The first processing step required to generate a sentence is enrichment by unification. This process consists in recursively unifying the input with the grammar as described in sections 2 and 3. Figure 6 shows the result of applying this step to the input from figure 5.

The second step is the linearization process, which extracts the lexemes and their features from the enriched structure shown in figure 6. The patterns from the enriched structure are processed with a pattern unifier to obtain a linear order for the lexemes. The result of this step is shown in figure 7. Afterwards, a simple morphological component is used to decline the lexemes and produce the final NL sentence.

5 Comparing the 2 systems

FUF's unifier and inference engine are composed of 27 Common Lisp modules, totaling around 9000 lines of code (about 300Kb).

```
gram(CL) :-
    given(CL:verb:lex),
    CL === [cat:clause,process_type:actions,verb:[process_class:actions]],
    CL:prot === (none ; [cat:np,animate:yes]),
    CL:goal === (none ; [cat:np]),
    CL:benef === (none ; [cat:np]),
    clause_voice(CL),
    transitive_class(CL),
    CL:verb:cat === verb_group,
    CL:subject:number === CL:verb:number,
    clause_pattern(CL).
```

becomes:

```
gram(_6988) :-
   unif([verb:[lex:_30289|_30302]|_30297],_6988),
   nonvar(_30289),
   unif(_6988,[cat:clause,process_type:actions,
               verb:[process_class:actions|_30338]|_30333]),
   gram_1(_6988),
   gram_2(_6988),
   gram_3(_6988),
   clause_voice(_6988),
   transitive_class(_6988),
   unif(_6988,[verb:[cat:verb_group,number:_30923|_57938],
               subject:[number:_30923|_30920]|_57929]),
   clause_pattern(_6988).

gram_1(_30505) :-
   unif([prot:none|_31087],_30505).
gram_1(_30472) :-
   unif([prot:[cat:np,animate:yes|_31123]|_31108],_30472).

gram_2(_30685) :-
   unif([goal:none|_31040],_30685).
gram_2(_30657) :-
   unif([goal:[cat:np|_31071]|_31061],_30657).

gram_3(_30865) :-
   unif([benef:none|_30993],_30865).
gram_3(_30837) :-
   unif([benef:[cat:np|_31024]|_31014],_30837).
```

Figure 3: Precompilation of a top-level rule

```
transitive_class(CL) :-
    CL:verb:transitive_class === Trans,
    (Trans = intransitive, CL === [object:none,iobject:none];
     Trans = transitive,   CL === [iobject:none];
     Trans = neutral,      CL === [iobject:none];
     Trans = bitransitive).
```

becomes:

```
transitive_class(_16911) :-
   unif([verb:[transitive_class:_16959|_53645]|_53640],_16911),
   gram_15(_16959,_16911).

gram_15(intransitive,_53968) :-
   unif(_53968,[object:none,iobject:none|_54019]).

gram_15(transitive,_53936) :-
   unif(_53936,[iobject:none|_54038]).

gram_15(neutral,_53904) :-
   unif(_53904,[iobject:none|_54057]).

gram_15(bitransitive,_53883).
```

Figure 4: Precompilation with indexation

```
[cat:clause,
 prot:[lex:john,np_type:proper],
 goal:[lex:ball,np_type:common,definite:no,
       describer:[lex:heavy]],
 benef:[lex:mary,np_type:proper],
 subject:[lex:mary,np_type:proper],
 verb:[dative_prep:to,transitive_class:bitransitive,
       lex:throw]]
```

Figure 5: Example input (structural form)

```
[cat:clause,
 prot:[lex:john,np_type:proper,cat:np,animate:yes,
       head:[lex:john,number:singular,cat:noun|_9433],
       number:singular,qualifier:none,
       pattern:[[head]|_9366],
       definite:yes,determiner:none,describer:none|_9503],
 goal:[lex:ball,np_type:common,definite:no,
       describer:[lex:heavy,cat:adj|_10099],cat:np,
       head:[lex:ball,number:singular,cat:noun|_9834],
       number:singular,qualifier:none,
       pattern:[[determiner,...,head,...],
                [...,describer,head,...]|_10067],
       determiner:[cat:det,definite:no,number:singular,
                   lex:a|_10184]|_9876],

                        ⋮

 pattern:[[subject,verb,...,object,...],
          [...,verb,object,by_obj,...]|_9139],
 iobject:none|_9186]
```

Figure 6: Enriched structure after unification

```
[[lex:mary,number:singular,cat:noun|_10401],
 [cat:verb,lex:be,number:singular|_11069],
 [cat:verb,ending:past_participle,lex:throw|_11150],
 ...,
 [cat:det,definite:no,number:singular,lex:a|_10184],
 ...,
 [lex:heavy,cat:adj|_10099],
 [lex:ball,number:singular,cat:noun|_9834],
 ...,
 [lex:by,cat:prep|_11291],
 [lex:john,number:singular,cat:noun|_9433],
 ...]
```

Figure 7: Linearized structure for final sentence “Mary is thrown a heavy ball by John”

A good part of this complexity comes from the fact that Lisp is not unification-based and does not perform automatic backtracking. The implementation of such a system in Prolog is much easier and produces a very compact engine, since most of the functionality is already present in Prolog itself.

Our prototype's inference engine is very small in comparison (about 250 lines of Prolog, which comes to around 5Kb). Add to that our small grammar pre-processor (less than 100 lines of Prolog) and you have most of the basic functionality of FUF in about one tenth of the code.

Using a language that is already rich in the areas of unification and backtracking instead of building on top of Lisp also gives us a net advantage when it comes to performance. Indeed, a good Prolog compiler is much more efficient at doing things than an implementation such as FUF.

To further improve on performance, we have decided to represent our rules as directly executable Prolog procedures, which eliminates a level of meta-interpretation that can really slow things down. This allows the Prolog compiler to do most of the optimizations for us.

Our system uses a pre-processor to translate grammars from the form shown in figure 2 into a series of direct calls to the feature unifier described in [Boy87]. Thus, the preprocessor eliminates the level of meta-interpretation associated with the expansion of the "`===`" operator's flexible syntax. It also translates the 2 types of disjunctions and the patterns allowed in our grammars, which [Boy87] doesn't address because it wasn't aimed at FUF-type grammars.

A series of performance tests using the FUF 5.1 examples associated with the grammar we used (including the input shown in section 4.2) show that our system is roughly 7–8 times faster than FUF at unification. This factor of improvement can really make the difference between a usable system and one that has too long a response time.

6 Conclusion

The prototype described in this paper has fulfilled all our expectancies. Indeed, it implements most of the important features of FUF while maintaining an astonishing level of simplicity. Its implementation remains very close to the Prolog language, which gives it a distinct advantage performance-wise.

Also, we have looked at other special features present in FUF (random alternatives, optional sub-structures, ...) and most of them can be added to the prototype with only a few lines of Prolog in the pre-processor (without changing the inference engine or the unifier). For example, random alternatives would be implemented in the preprocessor by generating calls to the `random` predicate in the Prolog code. Nevertheless, some meta-features such as

the keyword `any` require slight modifications in the unifier and the engine.

It would be possible to increase the system's performance by compromising a little at the versatility level. For example, it would be possible to completely eliminate the inference engine by letting the pre-processor issue all recursive calls directly in its output rules. These recursive calls would then be compiled along with the rest of the Prolog code for the rules and there would be no more need for an inference engine as the rules would do all the work themselves. Of course, since all of this would be done in the pre-processor, the grammars would stay unchanged and the user wouldn't have to enter the recursive calls by hand. This requires that the constituent set for all rules be known at compile time, and thus it probably requires that all possible inputs to the grammar be known in advance. It also forces a recompilation of the whole grammar to change the search algorithm.

Also, [ED86] shows how feature-list unification can be compiled into term unification. This transformation might also increase performance in many cases, but has not been tested in our framework. It is only feasible if all possible features in each rule are known in advance (at compile time) and do not depend on the input.

The PFUF system will be used in the framework of the SCRIPTUM project at the University of Montréal. This project unites a group of professors and graduate students interested in text generation. The language of choice in the group is Prolog and this was one of the motivations for the implementation of PFUF. Through its use by the various members of the project, this prototype implementation will evolve into a more complete system, gradually acquiring the more advanced features of FUF. Nevertheless, we will try to keep the system as small as possible, allowing anybody to modify it and integrate it into their research.

Acknowledgements

We wish to thank the Canadian Natural Sciences and Engineering Research Council for making this research possible through a scholarship. Michael Elhadad for his work on FUF and for making it publicly available. Leila Kosseim for helping with the translation of our example grammar from FUF to PFUF. Other members of the SCRIPTUM research group for their technical and moral support.

References

[Boy87] Michel Boyer. Towards Functional Logic Grammars. In P. St-Dizier, editor, *Proceedings of the Second International Workshop on Natural Language Understanding and Logic programming*, pages 46–62, Vancouver, 1987.

[DHRS92] R. Dale, E. Hovy, D. Rösner, and O. Stock, editors. *Aspects of Automated Natural Language Generation*. Springer-Verlag, 1992.

[ED86] Andreas Eisele and Jochen Dörre. A lexical functional grammar system in PROLOG. In *COLING-86*, 1986.

[Elh91] Michael Elhadad. FUF: the Universal Unifier. User Manual Version 5.0. Technical report, Columbia University, 1991.

[ER92] Michael Elhadad and Jacques Robin. Controlling content realization with functional unification grammars. In Dale et al. [DHRS92].

[GM89] Gerald Gazdar and Chris Mellish. *Natural Language Processing in PROLOG — An introduction to computational linguistics.* Addison-Wesley, 1989.

[Kay79] Martin Kay. Functional Grammar. In Christina Chiarello et al., editors, *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, pages 142–158, 1979.

[Kay85] Martin Kay. Unification in grammar. In Veronica Dahl and Patrick Saint-Dizier, editors, *Natural Language Understanding*, pages 223–240, Amsterdam, 1985.

[M+90] Kathleen R. McKeown et al. Language generation in COMET. In Mellish et al. [MDZ90].

[MDZ90] Chris Mellish, Robert Dale, and Michael Zock, editors. *Current Research in Language Generation.* Academic Press, 1990.

[MT90] Philip Miller and Thérèse Torris, editors. *Formalismes syntaxiques pour le traitement automatique du langage naturel.* Hermes, 1990.