Conditioned Unification for Natural Language Processing

Kôiti Hasida

Electrotechnical Laboratory
Umezono 1-1-4, Sakura-Mura, Niibari-Gun,
Ibaraki, 305 Japan

ABSTRACT

This paper presents what we call a conditioned unification, a new method of unification for processing natural languages. The key idea is to annotate the patterns with a certain sort of conditions, so that they carry abundant information. This method transmits information from one pattern to another more efficiently than procedure attachments, in which information contained in the procedure is embedded in the program rather than directly attached to patterns. Coupled with techniques in formal linguistics, moreover, conditioned unification serves most types of operations for natural language processing.

1. Introduction

A current major trend of natural language processing is characterized by the overall use of unification (Shieber (1984), Kay (1985), Proudin and Pollard (1985), Pereira (1985), Shieber (1985), etc.) reflecting the recent developments in nontransformational linguistic formalisms, such as Lexical Functional Grammar (Bresnan (1982)), Generalized Phrase Structure Grammar (GPSG) (Gazdar, Klein, Pullum and Sag (1985)), Head Grammar (Pollard (1984)), and Head-Driven Phrase Structure Grammar (HPSG) (Pollard (1985a,b)). These formalisms dispense with global operations such as transformation, and instead exploit local operations each confined within a local tree. Such local operations are formulated in terms of unification.

However, the ordinary unification as in Prolog is insufficient, seen from both scientific (here, alias linguistic) and engineering points of view. The problem is that patterns to be unified with each other lack the capacity for carrying information.

In this paper we present a new method of unification which we call **conditioned unification**. The essence of the method is to deal with patterns annotated by some sort of conditions. These conditions are so constrained as to be efficiently operated on, and yet to be able to carry rich enough information to capture linguistic generalizations.

2. The Problem

Ordinary patterns as in Prolog lack expressive power, because variables therein are simply indeterminate and thus carry almost no information. Therefore, such patterns and unification among them are insufficient for capturing the grammatical generalization and the processing efficiency. Let us look at some examples below. A grammatical category is assumed to be a list of features. A feature consists of a feature name and a value, and represented as a term like *name*(*value*).

The lexical entry of English verb *put*, for instance, cannot be described as a Prolog pattern, but needs some annotation (i.e., *put_tns_psn_nmb*(*T*, *P*, *N*)) as in (1).

(1) lexicon(put, [tense(T), person(P), number(N)]) :-
put_tns_psn_nmb(T, P, N).

Here, features other than *tense*, *person*, and *number* are omitted, and predicate *put_tns_psn_nmb* is defined as in (2).

(2) put_tns_psn_nmb(present, P, N) :-
not_3rd_sng(P, N).
put_tns_psn_nmb(T, P, N) :- not_pres(T).
not_3rd_sng(1st, N). not_pres(past).
not_3rd_sng(2nd, N). not_pres(past_participle).
not_3rd_sng(3rd, plural). not_pres(base).

For a bit more complicated instance, consider the relationship between a syntactic gap and its filler. In GPSG, HPSG, etc., this relationship is captured in terms of the SLASH feature, which represents gaps. In the category of *I think is crazy*, for example, the SLASH feature is specified as [NP]. Here SLASH is assumed to take as its value a list of categories. Stated below is a simplified principle about the distribution of this feature in typical cases.

(3) In a local tree, the mother category's SLASH feature is obtained by concatenating from left to right the SLASH features of her daughters.

In order to describe this principle, something more than a mere pattern is required again:

(4) local_tree([slash(X)], [slash(Y)], [slash(Z)]) :-
append(Y, Z, X).

Features other than SLASH are omitted here.

The so-called procedure attachments is the most common way of complementing the poor descriptive capacity of ordinary patterns. For instance, you may regard the bodies of Horn clauses (1) and (4) as attached procedures.

The drawback of procedure attachment is in the fact that the only way of using the procedures is to execute them. For this reason, procedures are merely embedded in programs, rather than attached to those patterns which these programs operate on. The information which procedures contain cannot generally be carried around across several partial structures each of which a procedure directly operates on, because, once a procedure is executed, the information which it contained is partially lost. For instance, when lexical entry (1) is exploited, *put_tns_psn_nmb*(*T*, *P*, *N*) is executed and *T* and *P* are instantiated to be *present* and *1st*, respectively. Thus left behind is the information about the other ways to instantiate those variables.

Actual procedure attachments must be arranged so that information should not be lost when procedures are executed. *Freeze* of Prolog (Colmerauer (1982)), for instance, is a means of this arrangement. By executing *freeze*(*X*, ψ), atomic formula ψ is frozen; i.e., the execution of ψ is suspended until variable *X* is instantiated. If ψ contains *X*, therefore, hopefully not so much information is lost when ψ is executed.

Nevertheless, *freeze* is problematic in two respects. First, information can still be lost when the frozen procedures are executed. Second, too much information can be accumulated while several procedures are frozen. Suppose, for instance, that *freeze*(*X*, *member*(*X*, [*a*, *b*])) and *freeze*(*Y*, *member*(*Y*, [*b*, *c*])) have been executed. Then, *X* and *Y* can be unified with each other without awakening either procedure. In that case, the information that *X* may be *b* is redundant between the two procedures, and the other part of information those procedures contain is inconsistent. What one might hope here is to instantiate *X* (and *Y*) to be *b*. If we had executed *freeze*(*Y*, *member*(*Y*, [*c*, *d*])) instead of *freeze*(*Y*, *member*(*Y*, [*b*, *c*])), computational

resources would be wasted as the price for a wrong processing.

After all, it is up to a programmer to take a deliberate care so that information should be efficiently transmitted across patterns. This causes several problems interwoven with one another. First, since those programs reflect the intended order of execution, they fail to straightforwardly capture the uniformities captured by rules or principles such as (3). Accordingly, programming takes much labor. Moreover, the resulting programs work efficiently only along the initially intended order.

3. Conditioned Unification

3.1. Conditioned Patterns

These problems will be settled if we can attach information to patterns, instead of attaching procedures to programs. Here we consider that such information is carried by some conditions on variables. Variables are then regarded as carrying some information rather than remaining simply indeterminate.

By a **conditioned pattern** let us refer to a pair of a pattern and a condition on the variables contained in that pattern. For simplicity, assume that the condition of a conditioned pattern consists of atomic formulas of Prolog whose argument positions are filled with variables appearing in the pattern, and that the predicates heading those atomic formulas are defined in terms of Horn clauses. For instance, we would like to regard the whole thing in (1) or (4) as a conditioned pattern.

3.2. Modular Conditions

The conditions in conditioned patterns must not be executed, or the contained information would be partially lost. The conditions have to be somehow joined when conditioned patterns are unified, so that the information they contain should be transmitted properly in the sense that the resulting condition is equivalent to the logical conjunction of the input conditions and contains neither redundant nor inconsistent information. We call such a unification a **conditioned unification**.

A simple way to reduce redundancy and inconsistency in a condition is to let each part of each possible value of each variable be subject to at most one constraint. Let us formulate this below. We say that a condition is **superficially modular**, when no variable appears twice in that condition. For instance, (5a) is a superficially modular condition, whereas (5b,c) are not. (Conditions are sometimes written as lists of atomic formulas.)

(5) a. [a(X, Y), b(Z), a(U, V)]
b. [a(X, Y), b(Y)]
c. [a(X, Y, X)]

Further we say that a condition Φ is **modular**, when all the relevant conditions are superficially modular. Here, the relevant conditions are Φ and the bodies of Horn clauses reached by descending along the definitions of the predicates appearing in Φ. A predicate is said to be **modular** when its definition contains only those Horn clauses whose bodies are modular conditions. A predicate is **potentially modular** when it is equivalent to some modular predicate.

A modular condition does not impose two constraints on any one part of any variable, and therefore contains neither redundancy nor inconsistency. Hereafter we consider that the condition in every conditioned pattern should be modular.

3.3. Expressive Power

Conditioned patterns can carry rich enough information for capturing the linguistic generality. Obviously, at first, they can describe any finite set of finite patterns. For instance, (1) is regarded as a conditioned pattern with modular condition [*put_ins_psn_nmb*(*T*, *P*, *N*)]. Moreover, also some recursive predicates are modular, as is demonstrated below.

(6) a. append([ ], Y, Y).
append([U | X], Y, [U | Z]) :– append(X, Y, Z).
b. sublist([ ], Y).
sublist([U | X], [U | Y]) :– sublist(X, Y).
sublist(X, [U | Y]) :– sublist(X, Y).

Thus, (4) is also a conditioned pattern.

However, some recursive predicates are not potentially modular. They include *reverse* (the binary predicate which is satisfied iff its two arguments are the reversals of each other, as in *reverse*([[*a*, *b*], *c*, *d*], [*d*, *c*, [*a*, *b*]])), *perm* (the binary predicate satisfied iff its arguments are permutations of each other, as in *perm*([1, 2, 3], [2, 1, 3])), *subset* (the binary predicate which obtains iff the first argument is a subset of the second, as in *subset*([*d*, *b*], [*a*, *b*, *c*, *d*])), etc.

Nevertheless, this causes no problem regarding natural language processing, because potentially infinite patterns come up only out of features such as SLASH, which do not require those non-modular predicates.

3.4. The Unifier

Shown below is a trace of the conditioned unification between conditioned patterns (7) and (8) (here we use the same notation for conditioned patterns as for Horn clauses), where the predicates therein have been defied as in (9). (The definitions of c0 and c3 are not exploited.) First, we unify [*X*, *Y*, *Z*, *W*] and [*A*, *B*, *C*, *D*] with one another and get *X* = *A*, *Y* = *B*, *Z* = *C*, and *W* = *D*. In the environment under this unification, the two conditions are concatenated, resulting in {c0(*X*), c1(*Y*, *Z*), c2(*Z*, *W*)}. The major task of this conditioned unification is to obtain a modular condition equivalent to this non-modular condition. This is the job of function *modularize*. *Modularize* calls function *integrate*, which returns an atomic formula equivalent to the given condition. The termination of a *modularize* or an *integrate* is indicated by ⇒ preceding the return-value, with the same amount of indentation as the outset of this function-call was indicated with. When an *integrate* calls a *modularize*, the alphabetic identifier of the exploited Horn clause is indicated to the left-hand side, and the temporal unification to the right-hand side. Atomic formulas made in *integrate* is written following ↓. Each Horn clause entered into the definition is shown following ↑, and given an alphabetic identifier indicated to the right-hand side.

(7) [X, Y, Z, W] :– c0(X), c1(Y, Z).
(8) [A, B, C, D] :– c2(C, D).
(9) c1(0, 1). (a)
c1(Q, 2). (b)
c2(1, P) :– c3(P). (c)
c2(2, 0). (d)

```
modularize([c0(X), c1(Y, Z), c2(Z, W)])
    integrate([c0(X)])
    ⇒ c0(X)
    integrate([c1(Y, Z), c2(Z, W)])
        ↓ c4(Y, Z, W)
    (a) modularize([c2(1, W)]) Y = 0, Z = 1
            integrate([c2(1, W)])
                ↓ c5(W)
            (c) modularize([c3(P)]) W = P
                    integrate([c3(P)])
                    ⇒ c3(P)
                ⇒ [c3(P)]
                ↑ c5(P) :– c3(P).          (i)
            ⇒ c5(W)
        ⇒ [c5(W)]
```

```
        ↑ c4(0, 1, W) :– c5(W).                    (j)
    (b) modularize([c2(2, W)]) Y = Q, Z = 2
            integrate([c2(2, W)])
                ↓ c6(W)
            (d) modularize([ ]) W = 0
                ⇒ [ ]
                ↑ c6(0).                           (k)
            ⇒ c6(W)
        ⇒ [c6(W)]
        ↑ c4(Q, 2, W) :– c6(W).                    (l)
    ⇒ c4(Y, Z, W)
⇒ [c0(X), c4(Y, Z, W)]
```

We can refine the program of *integrate* so that it should avoid any predicate whose definition contains only one Horn clause. For instance, the definition of $c5$ consists only of (i). Instead of (j), therefore, we may have $c4(0, 1, P) :– c3(P)$. Also (l) can be replaced by $c4(Q, 2, 0)$, based on (k).

We are able to work out recursive conditions from given recursive conditions. For example, consider how X and Z are unified under the condition (10), where *member* is defined as in (11).

```
(10)  [member(X, Y), c0(Z)]
(11)  member(A, [A | B]).                          (a)
      member(A, [C | B]) :– member(A, B).          (b)
```

The trace of this unification is shown below, where predicate $c1$ is recursively defined based on the recursive definition of *member*.

```
modularize([member(X, Y), c0(X)])
    integrate([member(X, Y), c0(X)])
        ↓ c1(X, Y)
    (a) modularize([c0(A)]) X = A, Y = [A | B]
            integrate([c0(A)])
            ⇒ c0(A)
        ⇒ [c0(A)]
        ↑ c1(A, [A | B]) :– c0(A).
    (b) modularize([member(A, B), c0(A)]) X = A, Y = [C | B]
            integrate([member(A, B), c0(A)])
            ⇒ c1(A, B)
        ⇒ [c1(A, B)]
        ↑ c1(A, [C | B]) :– c1(A, B).
    ⇒ c1(X, Y)
⇒ [c1(X, Y)]
```

It is a job of *integrate* to handle recursive definition. The last *integrate* above recognizes that the first *integrate*, which is trying to define $c1$, was called with the same arguments except the variable names. Hence the last *integrate* simply returns $c1(A, B)$, because the content of $c1$ is now being worked out under the first *integrate* and thus it is redundant for the last *integrate* to further examine $c1$.

It is not always possible for the above unifier to unify patterns under recursive conditions. For instance, it cannot unify X with Y under $[append(X, Y, Z)]$, because the resulting condition is not potentially modular. However, such a situation does not seem to occur in actual language processing.

4. Conclusion

We have presented a new method of unification, which we call a conditioned unification, where patterns to be unified are annotated by a certain sort of conditions on the variables which occur in those patterns. These conditions are so restricted that they contain as little redundancy as possible, and thus are always assured to be satisfiable.

This method has the following welcome characteristics. First, the patterns to be unified can carry abundant information represented by the conditions hanging on them. The expressive capacity of these conditions is sufficient for capturing linguistic generalities. Second, such information is effectively transmitted, by integrating the conditions when patterns are unified. Unlike procedure attachments, in this connection, the information-conveying efficiency of our conditioned unification is not affected by the direction of the data-flow. Therefore, our conditioned unification is completely reversible, and thus is promising as a tool for describing grammars for both sentence comprehension and production.

Owing to these characteristics, our conditioned unification provides a new programming paradigm for natural language processing, replacing procedure attachments which have traditionally enjoyed the ubiquity that they do not deserve.

References

Bresnan, J. (ed.) (1982) *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, Massachusetts.

Colmerauer, A. (1982) *Prolog II Reference Manual and Theoretical Model*, ERA CNRS 363, Groupe d'Intelligence Artificielle, Université de Marseille, Marseille.

Gazdar, G., E. Klein, G. K. Pullum, and I. A. Sag (1985) *Generalized Phrase Structure Grammar*, Basil Blackwell, Oxford.

Kay, M. (1985) "Parsing in Functional Unification Grammar," *Natural Language Parsing*, pp. 251-278, Cambridge University Press, Cambridge, England.

Pereira, F. C. N. (1985) "A Structure-Sharing Representation for Unification-Based Grammar Formalisms," *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, University of Chicago, Chicago, Illinois.

Pollard, C. J. (1984) *Generalized Phrase Structure Grammar, Head Grammars, and Natural Languages*, Doctoral dissertation, Stanford University, Stanford, California.

Pollard, C. J. (1985a) *Lecture Notes on Head-Driven Phrase-Structure Grammar*, Center for the Study of Language and Information,

Pollard, C. J. (1985b) "Phrase Structure Grammar without Metarules," *Proceedings of the Fourth West Coast Conference on Formal Linguistics*, University of Southern California, Los Angeles, California.

Proudin, D. and C. J. Pollard (1985) "Parsing Head-Driven Phrase Structure Grammar," *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, University of Chicago, Chicago, Illinois.

Shieber, S. M. (1984) "The Design of a Computer Language for Linguistic Information," *Proceedings of the 10th International Conference on Computational Linguistics*, Stanford University, Stanford, California.

Shieber, S. M. (1985) "Using restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms," *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, University of Chicago, Chicago, Illinois.