A Logic Programming View of Relational Morphology

Harvey Abramson
Institute of Industrial Science, University of Tokyo
Minato-ku, Roppongi 7-22-1, Tokyo 106, Japan
e-mail: harvey@godzilla.iis.u-tokyo.ac.jp

Abstract

We use the more abstract term "relational morphology" in place of the usual "two-level morphology" in order to emphasize an aspect of Koskenniemi's work which has been overlooked in favor of implementation issues using the finite state paradigm, namely, that a mathematical relation can be specified between the lexical and surface levels of a language. Relations, whether finite state or not, can be computed using any of several paradigms, and we present a logical reading of a notation for relational morphological rules (similar to that of Koskenniemi's) which can in fact be used to automatically generate Prolog program clauses. Like the finite state implementations, the relation can be computed in either direction, either from the surface to the lexical level, or vice versa. At the very least, this provides a morphological complement to logic grammars which deal mainly with syntax and semantics, in a programming environment which is more user-friendly than the finite state programming paradigm. The morphological rules often compile simply into unification of the arguments in the generated morphology predicate followed by a recursive call of the said predicate. Further speed can be obtained when a Prolog compiler, rather than an interpreter, is used for execution.

Introduction. Kimmo Koskenniemi's so called "two-level model" of computational morphology (1983) in which phonological rules are implemented as finite state transducers has been the subject of a great deal of attention. The two-level model is based partly on earlier work of Johnson (1972), who considered that a set of "simultaneous" phonological rules could be represented by such a transducer, and of Kaplan and Kay (1983) who thought that ordered generative rules could be implemented as a cascading sequence of such transducers. Koskenniemi in fact implemented the phonological rules by a set of finite state tranducers running in parallel, rather than by a single large finite state machine into which many cascading machines could be combined. Subsequent to Koskenniemi's original work, there was a LISP-based implementation called KIMMO (Kartunnen 1983), and two-level descriptions of English, Rumanian, French and Japanese (Kartunnen and Wittenburg, Khan, Lun, Sasaki Alam 1983). A later LISP based implementation by Dalrymple et al (1987) called DKIMMO/TWOL helped the user by converting two-level rules into finite state transducers: in earlier implementations, and in the recent PC-KIMMO system (Antworth 1990), it was the user's task to generate the machines from two-level descriptions.

However one very important contribution of Koskenniemi to morphology, namely the notion that there is a *relation* between the surface and lexical "levels", has been somewhat overlooked by implementation issues having to do with the conversion of two-level rules into finite state automata in the various KIMMO systems. The two-level rules according to this notion, unlike the rules of generative morphology which *transform* representations from one level to representations in the other, express a correspondence between lexical and surface levels. Furthermore since no directionality is implied in the definition of a relation, unlike generative rules, the same set of two-level rules applies both in going from surface to lexical levels and vice versa. Rather than being procedural rules, they are declarative. Consequently, any correct implementation of the two-level rules is a relational program which can be used either analytically or generatively. We will henceforth, in order to emphasize the fact that a relation is being defined by them, refer to relational morphology rules rather than to the mathematically neutral term "two-level rules".

Despite the recognition that relational morphology rules are declarative, the main emphasis in using them has been obscured by the original finite state implementation technique. Recently, Bear (1986) has interpreted such rules directly, using Prolog as an *implementation* language. This, although an improvement on finite state implementations from the point of view of debugging and clarity, still misses an important aspect of relational morphology rules as a declarative notation, namely that if relational morphology rules define a relation between surface and lexical levels, then that relation can be specified and implemented using any of several different relational programming paradigms. In this paper, we will show that logic programming, which can be viewed as a relational programming paradigm, can be used to give a declarative reading to morphological rules. Further, because of the execution model for logic programs, embodied in various logic programming languages such as Prolog, the declarative reading also has a convenient procedural reading. That is, each relational morphological rule may be thought of as corresponding to or generating a logic program clause. The entire set of logic program clauses generated from the relational morphological rules, coupled with some utility predicates, then constitutes a morphological analyser which can either be used as a stand alone program or which can be coupled as a module to other linguistic tools to build a natural language processing system. Since the rules have been transformed into logic program clauses, they gain in speed of execution over Bear's interpretive method, and further speed can be gained by compiling these clauses using existing Prolog compilers. At the very least, this provides a morphological complement to logic grammars (Abramson and Dahl 1989) which deal mainly with syntax and semantics, in a programming environment which we believe is more user-friendly than the finite state programming paradigm.

It may be argued that this is a step backwards from the linear efficiency of finite state processing. However, when discussing "efficiency" it is very important to be very precise as to where the efficiency lies and what it consists of. Finite state processing is linear in the sense that a *properly implemented* finite state machine will be able to decide whether a string of length n is acceptable or not in a time which is O(n), ie, for large enough n, a linear multiple of n. For small values of n, depending on how much bookkeeping has to be done, "finite state algorithms" may perform worse than algorithms which are formally $O(n^2)$ or higher. Any processing in addition to recognition may involve time factors which are more than linear. This entirely leaves aside the question of the user-friendliness of the finite state computing paradigm, a question of how "efficient" in human terms it is to use finite state methods. Anyone who has tried to implement finite state automata of substantial size directly (as in Koskenniemi's original implementation, the first KIMMO systems, and KIMMO-PC) will have realised that programming finite state machines is distastefully akin to directly programming Turing machines. A substantial amount of software is necessary in order to provide a development, debugging and maintenance environment for easy use of the finite state computing paradigm. There also remains the theoretical question as to the adequacy of finite state morphological descriptions for all, or even most, human languages. However, this is a topic we shall not venture into in this paper.

In our method, a relatively small Prolog program generates logic programming clauses from relational morphology rules. The generated clauses (at least in the experiments so far) are readable and it is easy to correlate the generated clause and the original morphological rule, thus promoting debugging. The standard debugging tools of Prolog systems (at the very least, sophisticated tracing facilities) seem sufficient to deal with rule debugging, and the readability of the generated clauses should help in the maintenance and transference of morphological programs. Thus, from the software engineering point of view, logic programming is a more sophisticated, higher-level programming paradigm than finite state methods. Also, should finite state descriptions in the end prove inadequate, or even inconvenient, for all of morphology, logic programming provides expressive power for reasonable extension of the notation of relational morphology rules. The current availability of Prolog compilers, even for small machines, provides another increment of speedy execution of the generated programs. Many of the morphological rules produce logic program clauses in which checking of the lexical and surface elements and contexts reduce to unification followed by a recursive call of the morphology predicate. Compiled Prolog abstract machine code for such clauses is usually very compact. Prolog compiler indexing mechanisms often make it possible to access the correct clause to be applied in constant time.

Notational Aspects. Our tableau notation for relational morphology rules is as follows:

```
LexLeft <= Lex => LexRight <:>
SurfaceLeft <= Surface => SurfaceRight
```

which expresses the relation between a lexical and a surface unit (Lex and Surface, respectively), provided that the left *and* right contexts at both the lexical and surface levels (LexLeft, LexRight, SurfaceLeft, and SurfaceRight) are satisfied.

Another kind of relational morphology rule which is allowed is:

```
LexLeft => Lex <= LexRight<:>
SurfaceLeft => Surface <= SurfaceRight.
```

which expresses a relationship between Lex and Surface providing that the left and right contexts at the lexical and surface levels are *different* from those specified by LexLeft, LexRight, SurfaceLeft, and SurfaceRight. This means that either LexLeft or LexRight is not satisfied, and also that either SurfaceLeft or SurfaceRight is not satisfied.

More compact notation is also accepted, for example:

```
LexLeft:SurfaceLeft
        <=Lex:Surface=>
                LexRight:SurfaceRight.

LexLeft:SurfaceLeft
        =>Lex:Surface<=
                LexRight:SurfaceRight.
```

In the case where a pair of lexical and surface contexts are identical, or if the lexical and surface elements are identical, they need not be repeated. Such compressed rules as the following are also allowed:

```
Left  <= Element => Right.
Left  => Element <= Right.
```

Sets of symbols may be specified: set(vowel,[a,e,i,o,u]).

Lexical entries may be specified as follows:

```
lexicon::{cat=noun,root= craps}.
lexicon::{cat=noun,root= piano}.
```

This feature notation is that used in the author's Definite Feature Grammars (Abramson 1991).

Logical reading of relational morphology rules. Corresponding to a set of relational morphology rules, a binary predicate morphology/2 specifies the relation between a lexical and a surface stream of characters:

```
morphology(LexStream,SurfaceStream).
```

In order to specify the logic program clause which corresponds to a relational morphology rule, we have to manipulate the left and right lexical and surface contexts. We can find the right contexts within LexStream and SurfaceStream, but we have to provide a specification of the left contexts, and we do this by defining the above binary predicate morphology/2 in terms of a quaternary predicate morphology/4:

```
morphology(LexStream,SurfaceStream,
                LeftLexStream,LeftSurfaceStream).
```

LeftLexStream and LeftSurfaceStream are initially empty and are represented as reverse order lists of the left contexts which have already been seen. The top level definition of morphology/2 is:

```
morphology(LexStream,SurfaceStream) :-
        morphology(LexStream,SurfaceStream,[],[]).
```

The fundamental logic program clause corresponding to a relational morphology rule such as

```
        LexLeft <= Lex => LexRight<:>
        SurfaceLeft <= Surface => SurfaceRight.
```

is

```
morphology(LexStream,SurfaceStream,
        LeftLexStream,LeftSurfaceStream) :-
 LexStream = [L1|LexRest],
 SurfaceStream = [S1|SurfaceRest],
 lexeme(Lex,L1), surface(Surface,S1),
 lex_context(LexLeft,LeftLexStream,LexRight,LexRest),
 surface_context(SurfaceLeft,LeftSurfaceStream,
        SurfaceRight,SurfaceRest),
 NewLeftLexStream = [L1|LeftLexStream],
 NewLeftSurfaceStream = [S1|LeftSurfaceStream],
 morphology(LexRest,SurfaceRest,
        NewLeftLexStream,NewLeftSurfaceStream).
```

Here, LexStream and SurfaceStream are analysed as consisting of the first characters L1 and S1, and the remaining streams, LexRest and SurfaceRest. It is verified that L1 is the lexeme specified by Lex, and S1 the surface character specified by Surface. (Lex, Surface, LexLeft, LexRight, SurfaceLeft, and SurfaceRight are as given in the morphology rule.) Contexts are checked by the subgoals lex_context and surface_context. New left context streams are created by prefacing the old left context streams with L1 and S1 (note again that the left context streams are built up in reverse order). Finally, the predicate morphology/4 is recursively called on the remainder of the lexical and surface streams, and with the new left context streams.

Although the logical reading of this appears to involve many subgoals, in fact for many relational morphology rules, the subgoals are compiled away into simple unifications. See the Appendix which contains a set of relational morphology rules dealing with simple English plural forms and their corresponding logic program clauses. Space does not permit us to comment on the example, or how the compiler works, but the interested reader may contact the author. Further papers will deal with these topics.

Conclusions and future research. We have provided here in the setting of logic programming a morphological complement to the logic grammars which mostly concentrate on syntax and semantics. However, we have also provided a notation and a logical reading of that notation which suggests further exploration as to expressiveness and efficiency. If the context in a relational rule is specified by a regular expression, the appropriate context stream is parsed using a small logic grammar which defines such expressions. It may however make sense, in approaching non-concatenative aspects of morphology, to be able to specify lexical contexts with more structure than regular expressions allow. The implementation would be easy: in place of the logic grammar used to parse regular expressions, a more complicated logic grammar (context free, at least) would be used for lexical context verification. It is also thought that metarules (see Abramson 1988 or Abramson and Dahl 1989) will be useful in dealing with nonconcatenative aspects of morphology. Since the project is at an early stage, there is not yet an extensive set of examples. We expect to develop a full set of rules for English morphology, and a specification of Japanese verb forms.

Acknowledgment. The author wishes to acknowledge the support of Ricoh Corporation as a Visiting Information Fusion Professor at the University of Tokyo's Institute of Industrial Science. The author also wishes to thank Prof. Akira Ishikawa of Sophia University, Tokyo, for comments on this work. Also, thanks to Profs. Ishizuka and Ishikawa for help in translating the abstract.

References.

Abramson, H. 1988. Metarules and an approach to conjunction in Definite Clause Translation Grammars. Proceedings of the Fifth International Conference and Symposium on Logic Programming. Kowalski, R.A. & Bowen, K.A. (editors), MIT Press, pp. 233-248, 1988.

Abramson, H. 1991. Definite Feature Grammars for Natural and Formal Languages: An Introduction to the Formalism. Natural Language Understanding and Logic Programming, III, edited by C.G. Brown and G. Koch, North-Holland, 1991.

Abramson, H. and Dahl, V. 1989. Logic Grammars, Symbolic Computation Series, Springer-Verlag.

Antworth, E.L. 1990. PC-KIMMO: A two-level processor for morphological analysis. Summer Institute of Linguistics, Dallas, Texas.

Bear, J. 1986. A morphological recognizer with syntactic and phonological rules. In Proceedings of COLING '86, 272-276. Association for Computational Linguistics.

Dalrymple, M. et al. 1987. DKIMMO/TWOL: a development environment for morphological analysis. Stanford, CA: Xerox PARC and CSLI.

Johnson, C.D. 1972. Formal aspects of phonological description. The Hague: Mouton.

Kaplan, R.M. and Kay, M. 1981. Phonological rules and finite state transducers. Paper presented at the 1981 Winter meeting of the ACL/USA.

Kartunnen, L. 1983. KIMMO: a general morphological processor. Texas Linguistic Forum 22:163-186.

Kartunnen, L. and Wittenburg, K. 1983. A two-level morphological analysis of English. Texas Linguistics Forum 22:217-228.

Khan, R. 1983. A two-level morphological analysis of Rumanian. Texas Linguistics Forum 22:253-270.

Koskenniemi, K. 1983. Two-level morphology: a general computational model for word-form recognition and production. Publication No. 11 Helsinki: University of Helsinki Department of General Linguistics.

Lun, S. 1983. A two-level morphological analysis of French. Texas Linguistics Forum 22:271-278.

Sasaki Alam, Y. 1983. A two-level morphological analysis of Japanese. Texas Linguistics Forum 22:229-252.

Appendix. Elementary formation of plurals in English.

```
(0)     x <= '+' => [s,'#'] <:>
        x <= e => [s,'#'].

(1)     z <= '+' => [s,'#'] <:>
        z <= e => [s,'#'].

(2)     y <= '+' => [s,'#'] <:>
         i  <= e => [s,'#'].

(3)     s <= '+' => [s,'#'] <:>
        s  <= e => [s,'#'].

(4)     o <= '+' => [s,'#'] <:>
        o <= e => [s,'#'].

(5)     [c,h] <= '+' => [s,'#'] <:>
        [c,h]  <= e => [s,'#'].

(6)                                   [s,h] <= '+' =>
[s,'#'] <:>
        [s,h]  <= e => [s,'#'].

(7)     _       <= y => '+' <:>
        in(con) <= i => _.

(8)     not(set([[c,h],s,[s,h],x,z,y])) <= '+' => [s,'#']
        <:>
        not(set([[c,h,e],[s,e],[s,h,e],[x,e],[z,e],[i,e]]))
        <=  0  => [s,'#'].

%Note negative context here.
(9)     set([[c,h],s,[s,h],x,z,y]) => e <= [s,'#'] <:>
        set([[c,h],s,[s,h],x,z,i]) => e <= [s,'#'].

%This is a default rule.
(10)    _ <= in(X,char_e) => _ <:>
        _ <= in(X,char_e) => _.

set(char_e,[a,b,c,d,f,g,h,i,j,k,l,m,
        n,o,p,q,r,s,t,u,v,w,x,y,z,'#']).
set(con,[b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,y,z]).
```

In addition to specifying characters such as s, x, etc., we can also define sequences of characters noted as lists [s,h], not(character), not(sequence of characters). in(con) means any member of the set con, whereas in(X,char_e) is a member of the set char_e assigned to the variable X for unification in another part of the rule. '+' is used as a morpheme boundary, 0 is used as the null symbol, '#' is used as an endmarker, and '_' is used to specify a don't care context. By providing a complete specification of context we could remove any consideration of ordering of the rules. However, it is convenient to depart slightly from an order free formalism by allowing default rules such as our last one with don't care contexts which specify what happens to symbols not dealt with in any of the afore-mentioned rules, to appear at the end.

```
(*)     morphology([], [], A, B).
(0)
morphology([+, s, #|A], [e, s, #|B], [x|C], [x|D]) :-
      morphology([s, #|A], [s, #|B], [+, x|C], [e, x|D]).
(1)
morphology([+, s, #|A], [e, s, #|B], [z|C], [z|D]) :-
      morphology([s, #|A], [s, #|B], [+, z|C], [e, z|D]).
(2)
morphology([+, s, #|A], [e, s, #|B], [y|C], [i|D]) :-
      morphology([s, #|A], [s, #|B], [+, y|C], [e, i|D]).
(3)
morphology([+, s, #|A], [e, s, #|B], [s|C], [s|D]) :-
      morphology([s, #|A], [s, #|B], [+, s|C], [e, s|D]).
(4)
morphology([+, s, #|A], [e, s, #|B], [o|C], [o|D]) :-
      morphology([s, #|A], [s, #|B], [+, o|C], [e, o|D]).
(5)
morphology([+, s, #|A], [e, s, #|B], [h, c|C], [h, c|D]) :-
      morphology([s, #|A], [s, #|B],
                 [+, h, c|C], [e, h, c|D]).
(6)
morphology([+, s, #|A], [e, s, #|B], [h, s|C], [h, s|D]) :-
      morphology([s, #|A], [s, #|B],
                 [+, h, s|C], [e, h, s|D]).
(7)
morphology([y, +|A], [i|B], C, [D|E]) :-
      con(D),morphology([+|A], B, [y|C], [i, D|E]).
(8)
morphology([+, s, #|A], [s, #|B], C, D) :-
      not substrings(C, [[h, c], s, [h, s], x, z, y]),
      not substrings(D, [[e, h, c], [e, s], [e, h, s], [e, x],
         [e, z], [e, i]]),
      morphology([s, #|A], [s, #|B], [+|C], D).
(9)
morphology([e, A, B|C], [e, D, E|F], G, H) :-
      not (substrings(G, [o, [h, c], s, [h, s], x, z, y]),
                 match([s, #], [A, B])),
      not (substrings(H, [o, [h, c], s, [h, s], x, z, i]),
                 match([s, #], [D, E])),
          morphology([A, B|C], [D, E|F], [e|G], [e|H]).
(10)
morphology([A|B], [A|C], D, E) :-
     char_e(A),
     morphology(B, C, [A|D], [A|E]).
```

Clause (*) is generated to terminate morphological processing when both the lexical and surface streams are empty. In this case, the left contexts are ignored.

Clauses 0-6 corresponding to rules 0-6 follow the same pattern in which lexical and surface symbols and contexts are specified within the streams and are checked by unification, followed by a recursive call to the morphology/4 predicate on the remainder of the lexical and surface streams and with new left context streams. Such clauses involving unification of the head and a body which is only a recursive call of the same predicate are efficiently handled by Prolog compilers. Clause 7 is similar except that it also involves a check to see that the first character in the left surface context is a consonant.

Set definitions such as:

```
set(con,[b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,y,z]).
```

generate unit clauses: con(b)., con(c)., ..., con(z).

In clause 7, if D in con(D) is instantiated (as it is since the D represents a left context which has already been seen), code generated for this by Prolog compilers amounts to something like an indexed table lookup which takes place in constant time. Similar remarks apply to clause 10 where it is checked that A is a member of the set char_e.

Clause 8 involves a combination of a unification check for the right context, and a check that the left context does not consist of any of the specified strings. Here, in order for the morphology clauses to work in both the analytical and generating directions, the negation must be logically safe notation, i.e., the arguments to the negation must be grounded. Logically safe negation involves the use of delaying evaluation of the negation until all arguments have been grounded.

Clause 9 which involves a negative context, makes sure, using safe negation, that either the right context is not [s,'#'] or that the left context does not match any of the substrings in the specified set.

Sample execution:

```
?- morphology(P,cries).
Root = cry                    %nonstandard plural formation
{cat=noun,                    %original lexicon entry
  root=cry}
Suffix = s
{cat=noun,                    %modified feature
  root=cry,
  plural=yes}
P = cry+s ;                   %another solution?
fail.                         %no.

?- morphology('fox+s',P).
P = foxes

?- morphology('piano+s',P).
P = pianoes;    %one plural form for some nouns
                %which end in "o"
P = pianos;     %another plural form
fail.           %no others
```

The same clauses for the predicates morphology/2 and morphology/4 are used in solving goals in both directions.

Japanese Summary.

関係型形態規則の論理プログラミングからの見方

概　要

Koskenniemiの研究による自然言語の語彙レベルと表層レベルの間に数学的関係を指定できるという視点は有限状態機械を使用した形態素解析では見過ごされてきたが、この点を強調するため本論文では通常の「二層形態論」のかわりに「関係的形態論」という更に抽象的な用語を用いる。有限状態機械あるいはそれ以外の場合でも、関係はいくつもある関係プログラミングのいずれの手法によっても計算できる。我々は（Koskenniemiのものに似た）関係的形態論規則の記法に論理的な読みを呈示する。これは実際にPrologのプログラム節を自動生成するのに使用できる。有限状態機械による実装の場合と似て、関係は表層から語彙、語彙から表層のいずれの方向でも計算できる。これによって少なくとも構文論及び意味論を主に扱う論理文法に形態論を補完することになり、それは有限状態パラダイムに較べてユーザー・インターフェースに優れたプログラミング環境を与える。件の形態素規則は多くの場合、生成された「形態論」述語の引き数の単一化とその述語の再帰的呼び出しに単純にコンパイルされる。実行にインタープリタでなくPrologコンパイラを用いれば、更に高速化が達成できる。

Harvey Abramson
〒106東京都港区六本木7-22-1
東京大学生産技術研究所