Natural Language Allomorphy in mo_lex

J. TROMMER
Universitaet Potsdam
Institut fuer Linguistik/Allgemeine Sprachwissenschaft Postfach 601553
D-14415 Potsdam
Tel: +49-331-9772936
e-mail: trommer@ling.uni-potsdam.de

Abstract

Linguistically, formalizations of natural language allomorphy should meet at least two adequacy criteria: The non-monotonic ("elsewhere") nature of allomorphy has to be captured and there must be a natural way to model phonological aspects of allomorphy. In this paper I present **mo_lex**, a formalism that meets both criteria by using violable and ranked finite-state constraints operating over strings of typed feature structures. Modifying an approach by Karttunen(1998) it is shown that the formalism is equivalent in generative power to finite-state transducers, and thus inherits the well-known formal properties of these devices, like computational tractability and a straightforward approach to parsing. It is superior in this respect to other approaches to allomorphy which are problematic both with respect to generative capacity and parsing.

1 Introduction

Morphemes in natural languages appear in different forms according to phonological and morphological contexts. Here, only the latter will be understood as allomorphy, a typical example being the past participle morpheme in English:

(1)

base form	**past participle**
shake	*shak-ed*
take	*take-en*

While it clearly depends on the stem, how *past participle* is realized, there is an asymmetry between *-en* and *-ed*, *-en* appearing only with a handful of stems, while *-ed* is the "regular" form occurring with the large majority of stems and with derived verb stems. A convenient formulation of the phenomenon is by ranked constraints, where each constraint has to be fulfilled only if this doesn't lead to the violation of a higher ranked constraint:

(2)

a. With *take,... past participle* is realized as *-en*.

b. *past participle* is realized as *-ed*.

In section 2 it is shown, how such constraints and hierarchies of them are implemented in **mo_lex**[1] finite-state transducers. Section 3 justifies and illustrates the use of feature structures in transducers. Some remarks on existing implementations of the formalism are given in 4. Finally (section 5) similarities and differences with other computational approaches to allomorphy are discussed.

2 Constraint hierarchies as finite-state transducers

A morphotactical component is assumed in form of a finite-state automaton, which characterizes possible word forms as strings of feature structures.[2]
For example the English past participle form of *take* would look like (3a). The task of **mo_lex** as a allomorphy component is then to supply each such string with a corresponding string of phonemical feature structures, depicted in (3b):
(3)[3]

[1] **mo** stands for "morphological", **lex** refers to UNIX-lex which has inspired the design of **mo_lex** in many respects.

[2] Cf. section 3 on the use of feature structures in finite-state machines.

[3] The feature content in the example is only meant as a sketch of a more complete analysis. Thus vowels are characterized only by [cons -] ("consonantal") without further differentiation. Consonants are further differentiated in coronals like *t* ([cor +]) and non-coronals like *k*. [nas +] characterizes nasals like *n*. The index-feature (*ind*) is assumed to distinguish lexical items from each other. *part* ("participle") is used as a morpheme category ("cat") value as is *v* ("verb").

a. [(cat v) (ind take)] [cat part]

b. [(cons +) (cor +)][cons -] [(cons +) (cor -)]
[cons -] [(cons +) (nas +)]

This is achieved in **mo_lex** by a sequence of allomorphic constraints of the form

(4)

(Left_Context\)Morph/(Right_Context) _ Phon

Morph stands for the part of the morphological string that has to be realized phonologically by *Phon*, *Left_Context* and *Right_Context* (optional) specify the contexts in which the constraints hold. *Morph* consists of a single feature structure. All other rule components can consist of arbitrary regular expressions over a finite alphabet of feature structures. (2) in this notation amounts to (5):

(5)

a. [ind take]\[cat part] _
[cons -] [(cons +)(nasal +)]

b. [cat part] _ [cons -][(cons +)(cor +)]

2.1 Semantics of single constraints

The syntax and semantics of single **mo_lex**-constraints (6a) corresponds quite closely to that of phonological rewrite rules (6b) which are interpreted as finite-state relations by Kaplan & Kay(1994) and Karttunen(1997).

(6)

a. (Left_Context\)Replaced(/Right_Context)
_ Replaces

b. Replaced _ Replaces
/(Left_Context\)___(/Right_Context)

In both cases parts of strings on a certain level have to correspond to ("to be replaced by") different strings in specified contexts. There is one substantial difference however: In rewrite rules those parts of input strings, that aren't replaced, correspond to identical segments in the output strings. In constraints on the morphology-phonology mapping this clearly is the wrong result. It should be simply left open to what phonemes morphemes not specified by a constraint correspond since this will be specified by other constraints. Ill thus take as a starting point Karttunens definition of his *Replace Operator*[4] and discuss only the modifications which have to be done to get the correct semantics for **mo_lex**-constraints. Karttunen implements his operator by the composition of 6 component relations 5 of which are necessary only for handling left and right contexts. For our purposes only the 6th one ("Replace"), which carries out the actual replacement has to be substantially modified. Its defined as

(7)[5]

(Id(ELSE)(REPLACED .x. REPLACES))*
Id(ELSE)

Id(X) is an operator that yields the (again regular) identity relation for a regular language X, i.e. the set of string pairs (S_1, S_2)from *X*, such that S_1 = S_2, while *X .x. Y* is the Cartesian product of the regular languages *X* and *Y*, i.e. the set of string pairs *(S_1, S_2)*, such that S_1 _ *X* and S_2 _ *Y*. *ELSE* abbreviates the set of strings not containing *REPLACED* namely ~$(REPLACED - ()).[6] To get the desired result for **mo_lex**-constraints (7) has to be replaced by:

(7´)

Any(ELSE)(REPLACED .x. REPLACES))*
Any(ELSE)

Any(X) is a regular relation which maps each string *S* from the regular language *X* to a string S_2 out of *[ ]** where |S_1| = |S_2|.[7]

Two further modifications of Karttunens algorithm are in order to guarantee that rules denote same-

[4] Actually the *Replace Operator* is parameterized w.r.t. the domain of context restrictions. In **mo_lex** Karttunens "Upward-Oriented Replacement" (1997: 129) is used, which matches closely the semantics rewrite rules in simultaneous rule application(Kaplan & Kay, 1994:347).

[5] The syntax for regular expressions used follows mainly the one of the UNIX-tool lex. Parentheses are used for grouping expressions. *(X |Y)* denotes the union of *X* and *Y*. Some operators not available in lex are adopted from Karttunen(1997) and explained below.

[6] Where "~" is the complement operator, "*$X* " denotes the string set containing at least one *X*, and "( )" is the set containing only the empty string. For details see Karttunen(1997).

[7] This is technically obtained by taking *DFA(X)* (i.e. the minimal deterministic finite-state automaton equivalent to *X*) and creating a finite-state transducer *FST* with the same states as *DFA(X)* and a transition Z_1 *-[ ]* _ Z_2 whenever *DFA(X)* has a transition Z_1 *-S_* Z_2.

length regular relations[8] which is crucial for the interpretation of constraint hierarchies in the next section. First, the *inner relation*, i.e. *(REPLACED .x. REPLACES)* in Karttunens formalization is allowed to consist of any regular relation. In **mo_lex** the *Morph* part of constraints (4) as the source of *REPLACED* in (7´) is restricted to single feature structures and the *inner relation* is extracted from the relation *Morph .x. Phon* in the following way: An equivalent finite-state transducer is constructed, and all _s in transitions of the form _/F and F/_ are replaced by a feature structure of a special type ("null") not used elsewhere in the constraints. The resulting transducer gives the needed *inner relation*.

Secondly each Context expression *C* has to be replaced by *C/Null* where Null is the designated null feature structure, and *X/Y* denotes *X* possibly interspersed with strings from *Y*. This has the effect that context specifications "ignore" the null symbols. [9]

2.2 Semantics of constraint hierarchies

For the implementation of constraint hierarchies a slightly modified version of an algorithm developed by Karttunen(1998) for the formalization of optimality theory (Prince & Smolensky, 1993) is used. [10] Assuming that there is no conflict between the single constraints the effect of the hierarchy could be implemented by the intersection of all constraints.[11]

The basic idea is now to make intersection "violable". The highest constraint in the hierarchy C_0 is intersected with the constraint that comes next in the hierarchy C_1. There will be morphological strings in *Left_Language*(C_0)[12], that have a phonological realization in the resulting transducer namely M_{in} = *Left_Language*(C_0 _ C_1), and others that do not, i.e. M_{out} = *Left_Language*(C_0) - *Left_Language*(C_0 _ C_1), where *X-Y* stands for the complement of *X* with respect to *Y*. *Violable_Intersection* of C_0 and C_1 is now defined as the union *Left_Restriction*(M_{out}, C_0) _ *Left_Restriction*(M_{in}, (C_0 _ C_1)) [13] The resulting transducer is again submitted to *Violable_Intersection* with the constraint coming next in the hierarchy and so on.

As an example take again our example of English past participle allomorphy. For the sake of simplicity the phonemes are replaced by strings, and the relation corresponding to (5a) is *left_restricted* to the simple morphotactics (8) in (9)[14]

(8)

([(cat v)(ind take)] |[(cat v) (ind shake)])
[cat part]

(9)

([(cat v) (ind take)]/[ ] [cat part]/en |
[(cat v) (ind shake)]/[ ] [cat part]/[ ])

Straightforward intersection with the transducer corresponding to (5b) gives

(10)

[(cat v) (ind shake)]/[ ] [cat part]/ed

Thus M_{in} is identical to the upper string of (10), while M_{out} is the upper string of

(11)

[(cat v) (ind take)]/[ ] [cat part]/en

(10) is *Left_Restriction*(M_{in}, *M*) and (11) is *Left_Restriction*(M_{out}, *M*), thus *Violable_Intersection(5a, 5b)* is their union:

(12)

([(cat v) (ind take)]/[ ] [cat part]/en |
[(cat v) (ind shake)]/[ ] [cat part]/ed)

[8] Same-length regular relations contain only string pairs (S_1, S_2), such that
|S_1| = |S_2| Cf. Kaplan & Kay,(1994:342).

[9] Similar techniques are used in Two-Level-Morphology, cf. Kaplan & Kay, 1994:367.

[10] The basic departure from Karttunen is the use of *FST*-intersection instead of composition. This is necessary since optimization in **mo_lex** is done according to "input-output"-constraints, which are equivalent to *FSTs*, while the original approach is restricted to "output"-constraints in form of *FSAs*.

[11] Note that finite-state transducers are not closed under intersection, but those of the same-length type are (Kaplan & Kay, 1994:342). This is the reason for introducing null symbols in the definition of **mo_lex**-constraints in the last section.

[12] *Left_Language(X)* is the range of *X*.

[13] *Left_Restriction(X, Y)* is the set of string pairs (S_1, S_2), such that S_1 is in the regular
language *X* and (S_1, S_2) is in the regular relation *Y*. It's defined as the composition *Id(X)* _ *Y*.

[14] Null feature structures are omitted.

3 Feature structures in finite-state machines

While it is a commonplace in the literature on finite-state machines to note that alphabets of finite-state machines can consist of finite feature structure sets (Johnson, 1972; Kaplan & Kay, 1994) without changing their generative capacities this possibility normally isn't used in practical applications.[15][16]

3.1 Formal treatment of feature structures

In fact it is possible to treat finite sets of fully specified feature structures simply as sets of atomic units. Under-specified structures are then abbreviations for sets. Thus in an alphabet with the features cons and back (each binary-valued) [cons +] denotes the set {[(cons +) (back +)], [(cons +) (back -)]}. This carries over trivially to regular expressions, while in finite-state automata transitions over a feature structures *F* abbreviate sets of transitions for all fully specified feature structures subsumed by *F*. While thus feature structures can be treated mathematically as simple symbols in practical applications its convenient to define certain notions and operations directly on feature structures. Some examples follow:

- A feature automaton accepts a string $S = F_{a1}.. F_{an}$ of feature structures iff there is a transition from the start state to an end state $T = F_{b1} ... F_{bn}$ and F_{ai} subsumes F_{bi} for 1 _ i _ n.
- In intersection of standard automata A_1, A_2, a transition from (Z_{1A1}, Z_{1A2}) to (Z_{2A1}, Z_{2A2}) over *Symbol* is added for every pair of transitions (T_1, T_2), where $T_1 = Z_{1A1}$- *Symbol* _ Z_{2A1} in A_1 and $T_2 = Z_{1A2}$- *Symbol* → Z_{2A2} in A_1. In feature automata (Z_{1A1}, Z_{1A2})- *F* _ (Z_{2A1}, Z_{2A2}) is added to the intersection automaton for every (T_1, T_2), where $T_1 = Z_{1A1}$- F_1 _ Z_{2A1} in A_1 and $T_2 = Z_{1A2}$-F_2 _ Z_{2A2} in A_1, and $F = Unify(F_1, F_2)$.
- For several purposes its necessary to convert nondeterministic feature automata in deterministic ones, which requires determining a set of successor states (possibly empty) for each state and each fully specified feature structure. Treating each such feature structure separately is avoided in the following way: For a state *S* and all types *T* such that there are no transitions of type *T* feature structures from *S* the successor set is empty. For all feature structures F of type *T* labeling transitions starting from *S* such that there's no other type *T* transition from *S* there are transitions for all type *T* feature structures specified for a single feature specified otherwise in *F* whose successor set is empty. All other transitions are split up in transitions of the fully specified feature structures subsumed by them. A similar approach is used in constructing the complement for sets of feature structures in computing the complement of automata.

3.2 Morphemes as feature structures

Its standard in the theoretical morphological literature (e.g. Halle & Marantz, 1993 and the references cited there) to formalize (abstract) morphemes as feature structures and to account in this way for natural syncretisms like in our somewhat modified example from English verbal inflection.[17]

(1´)

base form	**past tense**	**past participle**
shake	*shake-d*	*shak-ed*
prove	*prove-d*	*prove-n*

As is clear from these data *-d* isn't restricted to participle forms, but spells out any [past +] morpheme, i.e. [(cat fin) (past +)] and

[(cat part) (past +)]. (5) can thus be refor mulated as (5´):

(5´)

a.
[ind prove]\[(cat part) (past +)]
_ [cons -] [(cons +) (nasal +)]

b. [past +] _
[cons -][(cons +) (cor +)]

3.3 Partial allomorphy

Most allomorphy in natural language is partial in the sense that only single segments or features of single segments alternate: Nonproductive cases are the vowel alternations of English verbs as in *write,*

[15] See section 5 for some exceptions.

[16] Additionally to the apparatus developed by Kaplan & Kay **mo_lex** incorporates types, which restrict possible features and values.

[17] *Prove* is used here instead of *take* to avoid discussion of the classical problem, how to describe *took.* Cf. Hockett(1958)

wrote, written, a more regular variant of this can be found in Albanian:

(8)

Prs 3rd sg	**Pass 3rd sg**	**Past 1st sg**
shkrua-n	*shkru-het*	*Shkro-va*
"write"		
lye-n	*ly-het*	*Le-va*
"wash"		

The crucial point here is that the basic allomorphical pattern for the stems of *lyej* and *shkruaj* is identical. Both end in a [(high +) (round +)] [(high -) (round -)] vowel sequence in the 3rd person present active, in a single [(high +) (round +)] segment in the corresponding passive and in a single [(high -) (low -)] vowel in the past form. The difference between the verbs reduces to the fact that the stem of *shkruaj* ends in all cases with a sequence of [back +] that of *lyej* with [back -] vowels.

Surely a description not using phonological features cant capture adequately this pattern. In our formalism the following constraint hierarchy naturally accounts for the facts.[18]

(9)

[ind shkruaj] _ shkr[back +]*

[ind lyej] _ [back -]*

[cat v]/[tns past] _

[ ]*C[(high -) (low -)]

[cat v]/[voic pass] _

[ ]*C[(high +)(round +)]

[cat v] _

[ ]*C [(high +)(round +)]
[(high -) (round -)]

4 Implementation

The formalism has been implemented in a simplified form using the UNIX-tool flex which contains default applications of rules that are interpreted as finite-state automata. This implementation has been applied in a nearly complete formalization of the complex verbal inflection of Albanian, containing approximately 100 different conjugation patterns. However this implementation didn't use real feature structures, which means that partial allomorphy can be formulated only with the use of diacritics. No parsing procedure is available. A more recent implementation is based on a literal realization of feature structures and the algorithms described here in ANSI-C. It has been applied to small fragments of Amharic, English and Albanian.

5 Related work

Beesley(1998) shares the basic conception advanced here of allomorphy as the mapping of morpheme to phoneme strings, but he doesn't use feature structures or violable constraints. Work on finite-state phonology like Bird & Ellison(1994) formalizes phonological features in the form of auto-segmental tiers, but the implementation isn't directly comparable to the one presented here. DATR (Evans & Gazdar, 1996) allows analyses in much the same way as **mo_lex**, i.e. integrating non-monotonic reasoning and linguistically adequate representations (see especially Cahill, 1993), but DATR is formally equivalent to a Turing machine (Moser, 1992) and problematic w.r.t parsing. **mo_lex** thus offers a more restricted and efficient alternative.

References

BEESLEY, Ken (1998) *Arabic Morphology using only Finite-State Operations.* In: *Proceedings of the Workshop on Computational Approaches to Semitic Languages*, COLING-ACL ´98, Universitè de Montrèal.

BIRD, Steven & ELLISON, T.Mark (1994) *One level phonology: auto-segmental representations and rules as finite automata*. In: Computational Linguistics, 20,55-90.

CAHILL, Lynne J.(1993) *Morphonology in the Lexicon*. In: *Proceedings of the Fifth European Conference on Computational Linguistics*, 87-96.

EVANS, Roger & GAZDAR, Gerald (1996) *DATR: a language for lexical knowledge representation*. In: Computational Linguistics, 22, 167-216.

[18] For the sake of readability the starting feature structures of *shkruaj* and *lyej* are written as the corresponding segments. *C* abbreviates [cons +].

HALLE, Morris & Alec MARANTZ (1993) *Distributed Morphology and the Pieces of Inflection*. In: *The View from Building 20*, ed. Kenneth Hale and S.Jay Keyser. MIT Press, Cambridge, 111-176.

HOCKETT, C. (1958) *Two models of grammatical description*. In: Joos, M. (ed.)

Readings in Linguistics (2nd edn.) Chicago: University of Chicago Press.

JOHNSON, C. Douglas (1972) *Formal Aspects of Phonological Description*. The Hague.

KAPLAN, R. & KAY, M. (1994) *Regular models of phonological rule systems*. Computational Linguistics, 20(3), 331-378.

KARTTUNEN(1997) *The Replace Operator*. In: Roche, Emmanuel & Schabes, Yves (eds.) *Finite-State language Processing*, MIT Press.

KARTTUNEN, Lauri (1998) *The Proper Treatment of Optimality in Computational Phonology*. In: *Proceedings of the International Workshop on Finite-State Methods in Natural Language Processing*,Ankara, 1-12.

MOSER, Lionel (1992) *Simulating Turing Machines in DATR*. Brighton: University of Sussex, Cognitive Science Research paper CSRP 241..

PRINCE, A. & SMOLENSKY, P. (1993) *Optimality Theory: Constraint Interaction in Generative Grammar*. RuCCs TR-2, Rutgers University.