Finite-State Compilation of Feature Structures for Two-Level Morphology

François Barthélemy

CNAM (Cédric), Paris, France
INRIA (Atoll), Rocquencourt, France

Abstract. This paper describes a two-level formalism where feature structures are used in contextual rules. Whereas usual two-level grammars describe rational sets over symbol pairs, this new formalism uses tree structured regular expressions. They allow an explicit and precise definition of the scope of feature structures. A given surface form may be described using several feature structures. Feature unification is expressed in contextual rules using variables, like in a unification grammar. Grammars are compiled in finite state multi-tape transducers.

1 Introduction

Feature Structures are a convenient way of representing partial information. They have been broadly used for many purposes in Natural Language Processing.

Finite-State Morphology is an approach of computational morphology where the morphology of a natural language is described using contextual rules which denote a rational relation. These rules are simultaneous or sequential constraints. Each rule is compiled into a rational relation and all the relations are intersected or composed to obtain a unique relation implementing the grammar.

The use of feature structures for morphological computational descriptions is now very widespread. Many systems including Pc-Kimmo 2, Mmorph, Xerox Finite State Tool, have feature structures. Feature structure processing is usually performed by a separate engine, but there were some attempts to incorporate features into finite state machines.

Rémi Zajac [1] proposed a two-level formalism where the lexical level consists in a feature structure. This formalism is compiled into an extended finite-state transducer. Instead of concatenating the features of the lexical level, the extended finite-state machine unifies them. The surface representations are concatenated as usual. There is no restriction on the power of feature structure used, as far as the unification is decidable. The main drawback of the approach is that there is no other operation on feature structure than unification. The values may become more and more precise, but they can't change in a success path of the transducer. It is therefore impossible to encode informations such as the grammatical category that is changed by affix composition (e.g. `realization` where the form is a noun but the category of the root `realize` is verb).

More recently, Amtrup [2] proposed to use weighted finite state-machines, the weight being a feature structure. The idea is that feature structures with

union and unification are a semiring, which is the only property required on weights. Here again, there is no restriction on the feature structures, but the feature computation has to be monotonous. An important benefit with respect to Zajac is that there is no modification of the underlying finite-state machines.

Another approach is used by the Xerox tools [3]. There are no feature structure but *flag diacritics* which may be seen as independent features. There is a set of different operations on features: unification, but also positive setting, which gives a value to a flag, regardless of its previous value, negative setting which excludes a value for a given flag, positive and negative value test. All the operations act on a single feature, not a feature structure.

The flag operations are present in the finite state machines as special symbols concatenated to ordinary symbols. Operations are not performed by finite-state machines: they are performed at runtime, with an enumeration of the solutions and using a memory to store a single feature structure. Karttunen and Beesley propose to use features to decrease the size of finite-state machines. Finite-state machines over-generate, the over-generation being fixed at runtime. The system provides a command which transforms automatically a machine having flag diacritics into an equivalent machine without such flags, so the user may choose between run-time evaluation of these flags or compile-time evaluation, which may cause an explosion of the size of the machine.

Kiraz [4] proposed to compile features into finite-state automata. They are represented by strings of special symbols which are concatenated to the strings of grapheme/phoneme of the lexical level. In this framework, features are used only for two-level rule filtering: two-level rule application involves the unification of a feature structure associated to the rule with the feature structure associated to the lexical part of the feasible pair of the rule. Feature structures are always local to one morpheme. Features can't govern morpheme composition and no structure for the complete form is computed.

In this paper, we generalize the work of Kiraz in such a way that feature structures may be used not only for rule filtering, but also for affix concatenation. Instead of concatenating lexical representations and feature structure on the same level, they will be separated in two different levels. The rule application will involve possibly several feature structures. Two-level grammars become a kind of unification grammars. Such a formalism will allow the use of a feature structure as abstract representation of a form, following the proposition of Zajac.

Like Kiraz and the Xerox flag diacritics, some restrictions will apply on feature structures to allow their compilation as strings. Furthermore, some structural restrictions will apply on grammars in order to remain finite-state.

2 Compiling feature structures

In this section, we consider the compilation of a restricted kind of feature structures in strings. There is nothing new in this part of the paper: the techniques come from previous work about compilation of feature structures either in finite-state machines or in Prolog terms (see for instance [5]).

We first restrict ourselves to flat feature structures taking values in finite domains (small ones). Such structures are a convenient way of describing partial information, which is possibly incrementally enriched, by unification with other structures or by applying some rules.

A feature structure is a set of features, each feature being identified by its name and taking a value in a small domain. Each pair (name, value) may be implemented by a special symbol written <name=value>, and a feature structure by the concatenation of the symbols corresponding to its features. For instance, the features `pers` and `num` representing respectively the person and the number of an English verbal form, take respectively the values 1, 2, 3 and `sg` (singular), `pl` (plural). The alphabet used for implementing them are: <pers=1>, <pers=2>, <pers=3>, <num=sg> and <num=pl>. A structure `[num=sg,pers=1]` is represented by the string <num=sg><pers=1>.

To obtain the uniqueness of the representation, one has to use a fixed order between features such as for instance the lexicographic order between feature names. If one knows the set of features which may enrich a feature structure along the computations, a feature structure may be compiled into a regular expression implementing all these features. For instance, the structure `[pers=3]` is compiled into (<num=sg>|<num=pl>)<pers=3>.

Unifying two structures is equivalent to intersecting the strings representing them.

The compilation technique extends to embedded acyclic structures. The notion of feature name is just replaced by the notion of path. For instance:

$$\begin{bmatrix} \text{cat} & \text{name} \\ \text{agr} & \begin{bmatrix} \text{gender} & \text{masc} \\ \text{number} & \text{plural} \end{bmatrix} \end{bmatrix}$$

<agr.gender=masc><agr.number=plural><cat=name>.

Such an imbrication is convenient when several structures share the same substructure. This may be denoted using a single variable. In the compiled form, there will be no difference with respect to a flattened structure.

The disjunction and difference over regular expressions give support for feature structures with disjunctive and negative specification. For instance:

$$\begin{bmatrix} \text{person} & 1|2 \\ \text{tense} & \leftarrow \text{past} \end{bmatrix}$$

(<person=1>|<person=2>)(dom(tense)–<tense=past>)

3 Relating feature structures and strings

In the propositions of Zajac and Amtrup, a single feature structure is associated to respectively a surface form and a pair lexical and surface forms. For Kaplan

and Kay [6], there is a feature structure for each symbol, describing its phonological properties using binary features. Kiraz gives an example where a feature structure is associated to each lexical entry (typically, a morpheme), several such entries being concatenated to obtain a surface form. This means that morphological descriptions may use feature structures having different scopes with respect to the symbols of surface form. And why not using several types of structures with different scopes in the same description? For instance, one feature structure for each affix and another one for the complete form.

Partition-based morphology gives a way to implement this notion of scope. It is a variant of two-level morphology first defined by [7] and further improved by [8], [9] and [10]. Instead of describing a length preserving relation using symbol pairs, it uses pairs of strings of possibly different length. For instance an affix-based description of the form *impossibly* is (im:in)(possibil:possible)(ity:ity). In such a system, the pairing is not distributive with respect to concatenation, so the above string is considered different from (i:i)(mpossibil:npossible)(ity:ity), for instance. In other words, the splitting of strings in substrings is significant. In the implementations, the boundaries of substrings are represented using a special symbol. We will use the symbol `w`.

Such a segmentation of surface form is useful for feature structures. For instance, ([cat=name],spi)([number=plural],es). It is possible to use n-ary relations instead of binary relations, so the feature structures may be added to the two classical levels (lexical and surface) as a third level.

The compilation method proposed in [10] consists in compiling n-ary regular expressions in n-tape transducers synchronized on substrings terminators, inserted at the end of each pair – or tuple in the case of n-ary expressions. The other symbols are read independently, ordered according to the level they belong to. For instance, a string (aaa:xx)(b:yy) is compiled in the same-length expression a:0 a:0 a:0 0:x 0:x w:w b:0 0:y 0:y w:w, and then in the corresponding letter transducer.

The join operation is a way to merge relations (resp. transducers) which share some common components (resp. tapes). [10] shows that this operation is defined if the two operands have exactly one common level. This property holds even when this common level is split in two different ways in the two relations. A different substring terminator is used for each way.

We propose a multi-level formalism where regular expressions and contextual rules are extended to describe tree-structured relations. Each level in the tree is an n-tuple with n greater or equal to 1. The syntax `<i|` and `|i>` is used to respectively open and close a tuple at a depth `i` in the tree. `<0|` and `|0>` open and close the tuple at the root of the tree. Each member of the relation is composed of exactly one such tuple. `<0|` and `|0>` are used in the description as the string boundaries classically needed and sometimes written #.

Such structured representations are compiled using terminators, i.e. the tuple openings and commas separating their components disappears and the tuple closings are compiled into a special symbol ω_i read on all the relevant tapes.

[cat=name,num=pl]				
[type=root]				[type=suffix]
s	p	y		s
s	p	i	e	s

```
<0| [cat=name,num=pl],
    <1| [type=root],
        <2| s, s |2>
        <2| p, p |2>
        <2| y, i |2>
        <2| epsilon, e |2> |1>
    <1| [type=suffix],
        <2| s, s |2> |1> |0>
```

Fig. 1. An example of the tuple notation

In order to remain finite-state, tree-structures must be restricted. The syntax that we propose here refers explicitly to the depth of trees, so it describes depth-bounded trees, which are finite-state. A discussion of the tractable tree structures will take place in the last section of this paper.

We use a simplified version of the *generalized restriction rules* by Yli-Jyrä and Koskenniemi [11]. Let Π be a finite alphabet and $\diamond$ a symbol not in Π. A rule is written $W \Rightarrow W'$ where $W \subseteq \Pi^* \diamond \Pi^* \diamond \Pi^*$ and $W' \subseteq \Pi^* \diamond \Pi^* \diamond \Pi^*$. W is called the precondition, W' the postcondition. The diamonds are used to split strings in three parts: the left context, the center and the right context. Let $d_\diamond$ be the operator which deletes all the occurrences of the symbols $\diamond$ in a language. It may be formally defined as the composition with a finite transducer followed by a projection. The rule $W \Rightarrow W'$ denotes the language $\Pi^* - d_\diamond(W - W')$.

Informally speaking, if the precondition holds, then the postcondition has to be verified. The diamonds are markers inserted in regular expressions to define the center of the rule in such a way that precondition and postcondition apply on the same part of the strings. The context restriction and surface coercion rules from previous versions of two-level morphology may be written using this unique kind of rules.

In our system, the patterns W and W' of a rule $W \Rightarrow W'$ must be valid tree-structured regular expressions where the center is any part of the expression.

Feature structure types are declared as a set of names associated to finite domains, each value being a string. In the expressions, the features are explicitly typed. The type is given first, then the pairs (name, value). For instance, `[verb:pers=3,gen=m]` is a feature structure of type `verb`.

Variables may be used to represent a value shared by several features in an expression or in a rule. An expression with such a variable is equivalent to the disjunction of the expressions where the variable is replaced by a given value. Variables will be written with an identifier beginning with the symbol $.

4 Examples

In the first example, there is a unique feature structure associated to each form. This example consists in a partial description of the imperfective of the Arabic verb. The information about gender, number and person is given by prefix and suffix added to a core.

The description begins with some declarations. The type of feature structures is given as a list of feature names and for each name, the domain of values of the feature. In this first example, there are two levels of structure: there is a grouping of letters into affixes, and then a grouping of affixes into a form to which a feature structure is associated.

The morphotactics is defined using regular expressions. The construction `REGEXP` gives a name to the disjunction of regular expressions it contains. Each such expression is terminated by a semi-colon. The underscore stands for any adequately structured string (wildcard). It has different actual meanings depending on its context. The construction `LET` allows to define a regular expression by applying algebraic operations on previously defined expressions.

```
FEATURE TYPES
  verb: gen in {m,f}, pers in {1,2,3}, num in {sg,pl,du};
END TYPES
REGEXP prefix IS
  <0| [verb:pers=1,num=sg], <1| a |1> _ |0>;
  <0| [verb:pers=2], <1| t a |1> _ |0>;
  <0| [verb:pers=3,gen=m], <1| y a |1> _ |0>;
  ...
END
REGEXP core IS
  <0| [verb:_], <1|_|1> <1| k t u b |1> <1|_|1> |0>;
  ...
END
REGEXP suffix IS
  <0| [verb:pers=1|3], _ <1| epsilon |1> |0>;
  <0| [verb:pers=2,gen=f], _ <1| i i n a |1> |0>;
  ...
END
LET form=intersect(prefix,core,suffix);
```

The relation `form` obtained by intersection of the three descriptions of prefix, core and suffix, contains verbal forms such as for instance:

```
<0| [verb:pers=1,num=sg],
        <1| a |1><1| k t u b |1><1| epsilon |1> |0>
```

It is a structured representation of the form `aktub` (I write). The notation `epsilon` stands for the empty string. The description uses an empty suffix to describe cases where nothing is suffixed to the core.

In this example, the notion of circumfix is probably more relevant than prefix and suffix. In the proposed implementation, the coordination of prefix and suffix is obtained through feature structure unification. A more explicit alternative is to define directly the circumfixes using expressions such as:

```
<0| [verb:pers=2,num=pl,gen=f],
                <1| t a |1> _ <1| n a |1> |0>;
```

The above description gives the lexical form of a verb. To obtain the surface form, some phonological and graphemical rules are to be applied. It is possible to express them using a classical set of contextual rules which associates pairs of symbols from lexical and surface level, and then join this two-level system with the rational relation `form` defined here, identifying the lexical level of the two systems. The result of this join is a ternary relation.

The second example shows that it is possible to use feature structures not only for the morphotactics but in the contextual rules too. It consists in a partial description of French conjugation. There is a feature structure for each affix, having different types according to the affix. The sharp symbol in contextual rules is used to identify the center of the rule (instead of the $\diamond$ symbol used in the presentation of generalized restriction rules).

```
FEATURE TYPES
  root: conj in {1,2,3};
  suff1: tense in {pres, fut, past, cond, imp};
  suff2: conj, tense, pers in {1,2,3}, num in {sg, pl};
END TYPES
ABBREVIATION infix : for tuples depth 2;
REGEXP affix IS
  <1| [root:conj=1], a:_ i:_ m:_ |1>;
  <1| [root:conj=3], c:_ o:_ u:_ s:_ |1>;
  ...
  <1| [suff1:tense=pres], epsilon:_ |1>;
  <1| [suff1:tense=fut], R:_ |1>;
  ...
  <1| [suff2:tense=fut,pers=1,num=sg], a:_ i:_ |1>;
  <1| [suff2:tense=!passe,pers=1,num=pl], o:_ n:_ s:_|1>;
  ...
END
REGEXP morphotactics IS
  <0| <1| [root:conj=$C], _ |1><1| [suff1:tense=$T], _|1>
      <1| [suff2:conj=$C,tense=$T], _ |1> |0>;
END
LET affix_star=star(affix);
LET verbal_forms=intersect(morphotactics,affix_star);
RULES
  <0| <1| [root:conj=3], _ o:_ u:_ #s:_# |1>
    <1| [suff1:tense=fut], R:_ |1> _ |0> =>
```

```
  _ #s:d# _;
  <1| [root:conj=1], _ |1> <1| _, #epsilon:_ R:_# |1> _
            =>
  _ #epsilon:e R:r# _;
  ...
END
```

`$C` and `$T` are variables which represent any value for respectively the conjugation and the tense of a form. They are used to ensure that two feature structures have the same value, whatever it is.

Examples of forms are *aimerai* (I will love) and *coudrons* (we will sew):

```
 <0|   <1| [root:conj=1], a:a i:i m:m |1>
       <1| [suff1:tense=fut], epsilon:e R:r |1>
       <1| [suff2:tense=fut,pers=1,num=sg,conj=1],
                                      a:a i:i |1>
|0>;
<0|    <1| [root:conj=3], c:c o:o u:u s:d |1>
       <1| [suff1:tense=fut], R:r |1>
       <1| [suff2:tense=fut,pers=1,num=pl,conj=3],
                             o:o n:n s:s |1>  |0>;
```

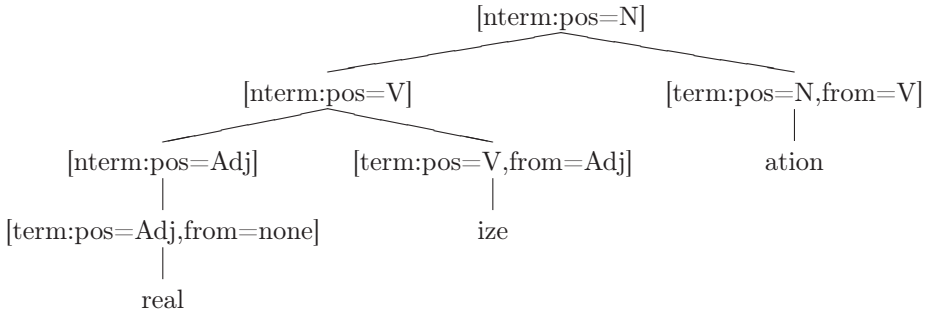
This example shows that no new construction is needed for using feature structures in contextual rules. The usual notions of context and center are sufficient. Features may be seen as a syntactic facility to express regular strings through a macro-expansion. The result of this macro-expansion is a set of regular generalized restriction rules which are compiled using the algorithm by Yli-Jyrä and Koskenniemi [11].

The third example illustrates how two different kinds of feature structures may be used in the same grammar: one kind will be devoted to the description of affixes; the other one to the composition of such affixes. They may be viewed as the terminal and non-terminal nodes of a unification grammar, respectively.

```
FEATURE TYPES
  term: pos in {N,Adj,V}, from in {N,Adj,V,none};
  nterm: pos;
END TYPES
CLASSES
  <letter>: a,b,c ...
END CLASSES
TUPLE TYPES
  <1| [nterm_], [term:_], <letter>* |1>;
END
REGEXP affix IS
  <1| _,[term:pos=Adj,from=none], real |1>;
  <1| _,[term:pos=V,from=none], move |1>;
  ...
```

```
  <1| _,[term:pos=V,from=Adj], ize |1>;
  <1| _,[term:pos=N,from=V], ation |1>;
  ...
END
LET affix_star=star(affix);
RULES suffixation ARE
<0| #<1| [nterm:pos=$P],_,_ |1># _ |0> =>
  _ #<1| [nterm:pos=$P],[term:pos=$P,from=none],_ |1># _;
_ #<1| [nterm:pos=$P1],_,_ |1># _ =>
  _ <1| [nterm:pos=$P2],_,_ |1> #<1| [nterm:pos=$P1],
                        [term:pos=$P1,from=$P2],_ |1># _;
END
LET stem=intersect(affix_star,suffixation);
```

This piece of code describes structures such as the following:



This syntax tree is encoded into a sequence of triples where each internal node is aligned with its rightmost child:

```
<0| <1| [nterm:pos=Adj],[term:pos=Adj,from=none],real |1>
    <1| [nterm:pos=V],[term:pos=V,from=Adj], ize |1>
    <1| [nterm:pos=N],[term:pos=N,from=V], ation |1>
|0>
```

The two generalized restriction rules are an encoding of the unification grammar:

$$\begin{bmatrix} \text{nterm} & \\ \text{pos} & \boxed{1} \end{bmatrix} \rightarrow \begin{bmatrix} \text{term} & \\ \text{pos} & \boxed{1} \\ \text{from} & \text{none} \end{bmatrix}$$

$$\begin{bmatrix} \text{nterm} & \\ \text{pos} & \boxed{1} \end{bmatrix} \rightarrow \begin{bmatrix} \text{nterm} & \\ \text{pos} & \boxed{2} \end{bmatrix} \begin{bmatrix} \text{term} & \\ \text{pos} & \boxed{1} \\ \text{from} & \boxed{2} \end{bmatrix}$$

Let us detail the compilation of the simplest of the two rules:

```
<0| #<1| [nterm:pos=$P],_,_ |1># _ |0> =>
  _ #<1| [nterm:pos=$P],[term:pos=$P,from=none],_ |1># _;
```

The first step consists in replacing the variable appearing in the center of the rule, namely `$P`, by its possible values, resulting in a set of three rules:

```
<0| #<1| [nterm:pos=N],_,_ |1># _ |0> =>
  _ #<1| [nterm:pos=N],[term:pos=N,from=none],_ |1># _;
<0| #<1| [nterm:pos=Adj],_,_ |1># _ |0> =>
  _ #<1| [nterm:pos=Adj],[term:pos=Adj,from=none],_ |1>#
  _;
<0| #<1| [nterm:pos=V],_,_ |1># _ |0> =>
  _ #<1| [nterm:pos=V],[term:pos=V,from=none],_ |1># _;
```

The second step replaces each feature structure by its compiled form as a symbol string. The wildcard symbol is replaced by the relevant expression where the symbol `<any>` stands for any regular symbol in the alphabet (all the symbols except the end of tuple `<wi>` and #). For the first of the three rules from the previous step, it gives:

```
<0| #<1| <nterm><pos=N>,<any>*, <any>* |1>#
    <1| <any>*,<any>*,<any>*  |1>* |0> =>
 <0| <1| <any>*,<any>*,<any>*  |1>*
     #<1| <nterm><pos=N>,<term><pos=N><from=none>,<any>*
      |1>#
     <1| <any>*,<any>*,<any>* |1>* |0>;
```

The third step consists in compiling the tuples using the techniques presented in the section 3. Like in classical Two-Level morphology [12], 0 is a special symbol inserted to obtain same-length relations. It is treated alternatively as an ordinary symbol (for intersection) or as the empty string (for composition).

```
#:#:# <nterm>:0:0 <pos=N>:0:0 (0:<any>:0)* (0:0:<any>)*
<w1>:<w1>:<w1> #:#:#
((<any>:0:0)* (0:<any>:0)* (0:0:<any>)* <w1>:<w1>:<w1>)*
          =>
#:#:# <nterm>:0:0 <pos=N>:0:0 0:<term>:0 0:<pos=N>:0
      0:<from=none>:0 (0:0:<any>)* <w1>:<w1>:<w1> #:#:#
((<any>:0:0)* (0:<any>:0)* (0:0:<any>)* <w1>:<w1>:<w1>)*
```

Finally, the rule is compiled using the formula from [11], namely $\Pi^* - d_\diamond(W - W')$ where W and W' are respectively the left-hand side and the right-hand side of the rule and Π^* the support of the relation. In our example,
Π^* is a sequence of triples `<1|_,_,_|1>*`. Thanks to the type declaration of the tuples, the compiler knows that it is more precisely:
`<1| [nterm_], [term:_], <letter>* |1>*`, which compiles into the following:

```
(<nterm>:0:0 (<pos=V>:0:0|<pos=N>:0:0|<pos=Adj:0:0>)
0:<term>:0 (0:<pos=V>:0|0:<pos=N>:0|0:<pos=Adj:0>)
(0:<from=none>:0|0:<from=V>:0|0:<from=N>:0|
                                    0:<from=Adj:0>)
(0:0:<letter>)* <w1>:<w1>:<w1>)*
```

The result of the compilation of the rule is given in the figure 2. The notation `<name=_>` is used as an abbreviation which stands for any symbol associating a value to the feature `name`.

5 Theoretical and practical limits

There are two kinds of limits to the compilation of feature structures using tree-structured relations: theoretical limits due to the kind of tree structures which can be represented in finite-state machines; practical limits due to the size of the finite-state machines.

Not all structure are implementable as finite-state machines. It is well-known, for instance, that context-free parsing is not finite-state. Chomsky in [13] gives a characterization of grammars which are regular. A grammar is said *self embedding* if there exists a derivation $A \stackrel{*}{\rightarrow} \alpha A \beta$ where A is a non-terminal and α and β are non-empty strings. A grammar is regular if and only if it is not self-embedding. This includes finite, right-linear and left-linear grammars.

Note that our examples use implicitly linear structures although it seemingly describes only finite structures because sequences of tuples of a given level are allowed within a tuple of higher level. For example in
`<0| [verb:_], <1|_|1> <1| k t u b |1> <1|_|1> |0>`, there is a sequence of three tuples of depth 1 as second component of the tuple of depth 0.

Linear structures are sufficient to express some morphologies, such as for, instance, Turkish morphology or French flexion which use mostly suffixes. They are not sufficient to represent English or French derivation which use both prefixes and suffixes. The solution in these cases are to restrict to use only finite grammars, for instance by limiting the depth of recursion for self-embedding non-terminals.

From a practical point of view, descriptions involving tree-structured relations may be too large to be compiled and executed. Feature structures may describe long-distance dependencies like in the example of circumfixation of Arabic verbal forms. We have implemented a prototype which converts the formalism presented in this paper into genuine finite-state automata and uses the FSM toolkit [14] to compile and execute them. We have written a number of sample grammars for French and Turkish verbs and a medium-size grammar of the Akkadian verb (about 50 rules). During these experiments, we sometimes encountered size explosion that we resolved by a careful writing of grammars and ordering of algebraic operations

Feature structures must be limited to a small number of features having small domains. Like feature diacritics in Xerox Tools, feature structures in our system could be evaluated at run-time, when a composition with a surface or abstract form drastically decreases the size of the machine. Instead of performing unification at compile-time, equations giving values to features should be concatenated within each scope, i.e. in each tuple.

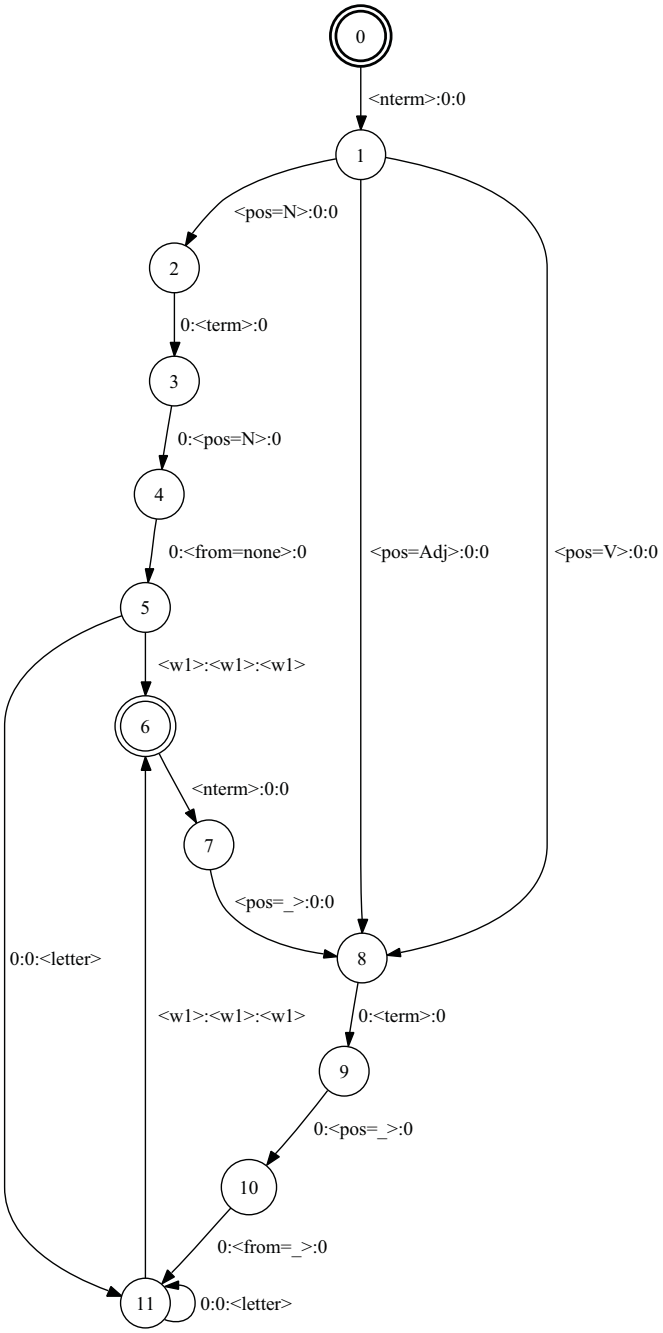


Fig. 2. Result of the rule compilation

6 Conclusion

The technique proposed in this paper is an improvement of the proposition by Kiraz, namely compiling feature structures into regular expressions which are part of a n-ary relation. The benefits of our approach are a more flexible use of the features and the possible simultaneous use of several kinds of scope for feature structures. There is also a better integration to contextual rules: the feature structures are part of the contexts and centers, and not a side condition.

With respect to the propositions by Amtrup and Zajac, the compilation in regular expressions offers a better integration into the two-level formalism. On the other hand, there are restrictions on the kind of structures and a risk of explosion of the size of the machines.

Ideally, a smart compiler should analyze grammars using unrestricted feature structures and automatically separate them in three parts: a small number of features or features approximations which are statically compiled, a second set of features which are computed at runtime as a finite-state operation (e.g. transducer composition), after the composition with a surface (or abstract) form and finally the features which are not computable using finite-state machines, and which would be evaluated separately for each solution by an external device. There is still a lot of work to perform such a statical analysis of grammars and to improve compilation techniques for the first two subsets of features.

References

1. Zajac, R.: Feature structures, unification and finite-state transducers. In: FSMNLP'98: International Workshop, on Finite State Methods in Natural Language Processing. (1998)
2. Amtrup, J.W.: Morphology in machine translation systems: Efficient integration of finite state transducers and feature structure descriptions. Machine Translation **18**(3) (2003) 217–238
3. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI Publications (2003)
4. Kiraz, G.A.: Compiling regular formalisms with rule features into finite-state automata. In: ACL, Madrid, Spain (1997)
5. Shöter, A.: Compiling feature structures into terms: an empirical study in prolog. Technical Report EUCCS-RP-1993-1, ICCS, Edinburgh, Scotland (1993)
6. Kaplan, R.M., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20:3** (1994) 331–378
7. Black, A., Ritchie, G., Pulman, S., Russell, G.: Formalisms for morphographemic description. In: Proceedings of the third conference on European chapter of the Association for Computational Linguistics (EACL). (1987) 11–18
8. Pulman, S.G., Hepple, M.R.: A feature-based formalism for two-level phonology. Computer Speech and Language **7** (1993) 333–358
9. Grimley-Evans, E., Kiraz, G., Pulman, S.: Compiling a partition-based two-level formalism. In: COLING, Copenhagen, Denmark (1996) 454–459
10. Barthélemy, F.: Using Mazurkiewicz trace languages for partition-based morphology. In: ACL, Prague (Czech Republic) (2007)

11. Yli-Jyrä, A., Koskenniemi, K.: Compiling contextual restrictions on strings into finite-state automata. In Watson, B., Cleophas, L., eds.: Proc. Eindhoven FASTAR Days, Eindhoven, Netherlands (2004)
12. Koskenniemi, K.: Two-level morphology: a general computational model for word-form recognition and production. Technical Report 11, Department of General Linguistics, University of Helsinki (1983)
13. Chomsky, N.: On certain formal properties of grammars. Information and Control **2**(2) (1959) 137–167
14. Mohri, M., Pereira, F.C.N., Riley, M.: Weighted finite-state transducers in speech recognition. Computer Speech and Language **16**(1) (2002) 69–88