FTrace: a Tool for Finite-State Morphology

James Kilbury
Heinrich-Heine-Universität
Düsseldorf
kilbury@phil.uni-duesseldorf.de

Katina Bontcheva
Heinrich-Heine-Universität
Düsseldorf
bontcheva@phil.uni-duesseldorf.de

Younes Samih
Heinrich-Heine-Universität
Düsseldorf
samih@phil.uni-duesseldorf.de

Abstract

In this paper we describe our work in progress on FTrace, a tool for finite-state morphology that provides a tracing facility for developers of applications for synchronic and diachronic language descriptions. We discuss not only the current tool for downward tracing, but also the challenges that we face in the further development of FTrace, especially in upward tracing. Finally, we present an example, draw some conclusions, and outline our future work. **Keywords:** FTrace, tracing, finite-state morphology, xfst, foma, SWI Prolog, Prolog network-interpreter, diachronic language descrip-tion.

1 Introduction

In this paper we describe FTrace, a tool for finite-state morphology that provides a tracing facility for developers of applications for synchronic and diachronic language descriptions. It constitutes work in progress, and we therefore will discuss not only the current tool for downward tracing, but also the challenges that we face in the further development of FTrace, especially in upward tracing.

In Section 2 we explain our motivation for creating FTrace, in Section 3 we describe the overall architecture of the system and outline briefly the language-description module and the visualization environment. In Section 4 we describe in detail the module for compiling and exporting the individual networks for replacement rules into Prolog notation before we describe the Prolog network-interpreter and the specific problems we face in downward and upward tracing in Section 5. Finally, in Section 6 we illustrate the tool with an example, and in Section 7 we draw some conclusions and outline our future work.

2 Motivation

In the past decade software systems like xfst and foma (cf. Beesley and Karttunen, 2003; Hulden, 2009, respectively) based on finite-state technology have greatly facilitated the use of replacement rules (Karttunen, 1995) in computational descriptions of natural languages. The vast majority of such applications have been purely synchronic and often employ replacement rules to capture morphophonemic alternations within lexical paradigms.

The replacement format, however, makes the rules equally attractive as a framework in which the phonology of a language can be modelled diachronically. The formal process is essentially the same, whether we synchronically derive surface forms from underlying lexical representations with morphophonemic rules, or diachronically derive later representations from corresponding earlier forms. In both cases we have a binary relation consisting of pairs $<w_0, w_n>$ of an upper-level string w_0 and a lower-level string w_n defined in the description by a sequence of replacement rules $R_1, \ldots, R_n$ and implicitly by the corresponding derivation $w_0, w_1, \ldots, w_n$, where each string w_j for $j \geq 1$ is produced from string w_{j-1} by the application of R_j.

So much for the elementary formal language theory. The whole point of systems like xfst is that the entire sequence $R_1, \ldots, R_n$ of replacement rules is composed and compiled into a single network encoding the binary relation. So we don't see any of the intermediate strings w_j of the derivation, and

Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing, pages 88–92,
Blois (France), July 12-15, 2011.

that is precisely what makes finite-state technology so efficient.

Once an application has been correctly developed, we normally have no need or desire to see derivations, but the situation is different if we have difficulty formulating replacement rules in a way that gives us the results we want. This is especially the case for students learning to use xfst or foma to encode linguistic descriptions. Then it is very helpful to be able to follow entire derivations in terms of the individual rule applications. This is just what a tracing facility would provide, but neither xfst nor foma has one. Vi-xfst (cf. Oflazer and Yilmaz, 2004) has a very useful dependency-tracking tool that we can use to visualize relationships between rules, but it includes no tracing facility that meets our requirements.

A first glance this appears to be a job that the developers of xfst or foma should do in order to produce the best tracing tool, but there is a short cut that requires no changes in the source code or involvement of the original developers. A feature of both systems is that they allow individual networks to be exported in Prolog notation. If each rule R_j is exported as a corresponding network N_j, then it is easy for a user to write his own Prolog program to interpret the individual networks as governed by a "play list" of the names of rule networks in their order of application. Such Prolog programming is described, e.g., in (Gazdar and Mellish, 1989, p. 37 ff.) and serves as the basis for our Prolog code which we specify below in 5.2 for the benefit of readers unfamiliar with (Gazdar and Mellish, 1989).

In our own work we have adopted the latter strategy.

3 Architecture of the Tool

There are four modules in FTrace:

1. the language-description module,
2. the module for export of xfst/foma networks in Prolog notation,
3. the Prolog interpreter,
4. the visualisation environment (SWI Prolog).

3.1 The Language-Description Module

The language-description module can consist of a single xfst script that contains the (continuation) lexicons and the replacement rules as well as variable declarations that define natural classes of segments and clusters of morphological tags. However, it can be a full-fledged lexicon that contains an xfst script with the variable declarations, the replacement rules, etc., a lexc master lexicon, and several (lexicographic) text files that contain the stems belonging to different inflectional classes.

It is very important to mention that the xfst script must comply with a number of special conventions.

Since we are not interested in the individual networks of tag clusters or natural classes of segments but want to export only the individual networks of the replacement rules, we need to introduce different naming conventions. Thus, the names of tag clusters or natural classes of segments begin with a capital letter (e.g., **VowFrt**, **Vowfrt**, **TAGS**), while the names of the rules begin with a lower-case letter (e.g., **r2**, **jerFrtVoc**).

The second convention refers to the play list that is needed by the Prolog network-interpreter (cf. above, Section 2, Motivation). The play list can be a part of the language-description module and have the form of a regular expression that denotes a cascade of replacement rules and has xfst syntax. It is possible, however, for the play list to be omitted from the language-description module. In this case the developer writes the play list as an xfst comment. In both cases the name of the regular expression must begin with a lower-case letter and the line must contain an xfst comment '# … **QQ**', which marks the play list for Perl (cf. Section 4). We have chosen this particular string because it is highly unlikely to be a substring of any reserved or natural-language word.

3.2 The XFST/Foma-Specific Module(s)

Of the four modules only the export module is specific to either xfst or foma. This is necessary since there are some important differences between xfst and foma:

- an xfst script cannot be started from outside the application environment but can make calls to the system;
- a foma script can be started from outside the application environment but cannot make calls to the system (last tested version: 0.9.14alpha).

In Section 4 we explain in detail an export module for xfst.

3.3 The Visualisation Environment

Since the Prolog network-interpreter will be described in detail in Section 5, we still need to say a few words about the visualisation environment. Our interpreter is compatible with most distributions of Prolog. However, we have chosen SWI Prolog for the following reasons:

- SWI Prolog is widely used for research and in instruction, it offers a comprehensive environment and is free;
- It is very easy to type UTF8 characters in the console and to display them. This makes SWI Prolog an ideal environment for tracing with language descriptions that use various writing systems such as, e.g., Cyrillic and Arabic.

4 The Module for Compiling and Exporting the Prolog Networks

The module consists of several xfst and Perl files. The task is to print the Prolog networks for each replacement rule and to export the rules to a separate Prolog UTF8 file that will be used by the network interpreter. In addition, the play list that is specified in the language description has to be extracted and added to that file. All tasks in this module run automatically; the user just needs to provide the name of the language-description file.

The main element of this module is an xfst script *ftrace.xfst*. First it starts an interactive Perl script *GetName.perl.pl*[1] that asks the user to type in the name of the language description file and saves it to a text file[2] *LDSrc.txt*. In the next steps the language description file is compiled, and the names of the defined variables are printed and saved to a file *defined.txt*.

The second Perl script *PrintNwks.perl.pl* selectively extracts from *defined.txt* only the names of variables that (according to the convention) begin with a lower-case letter, and dynamically creates an xfst script *print-prolog-source.xfst* that prints the corresponding Prolog networks. However, the names that are assigned to the networks by xfst have nothing to do with the names given to the rules by the linguist. The original names of the rules are restored automatically before the networks are saved to a Prolog file *<filenameoflangdescr>-Nwks.pl*.

The third Perl script *PrintPlList.perl.pl* extracts the play list and rewrites it in Prolog syntax. Then the play list is appended to the file (*<filenameoflangdescr>-Nwks.pl*) that contains the Prolog networks.

Finally, the Perl script reminds the user that his files are saved to a temporary directory and will be deleted with the next execution of FTrace. The name and the location of the file that contains the Prolog networks and the play list are also displayed.

5 The Prolog Network-Interpreter

The complexity of the tracing interpreter depends chiefly on the sublanguage of xfst or foma we wish to cover for tracing. For the time being we have excluded special features such as flag diacritics and merge in xfst (cf. Beesley and Karttunen, 2003: 339 ff., 401 ff.) and assume simply (1) variable definitions to specify natural segment classes, (2) elementary regular expressions to define the distribution of segments in the upper-level language (i.e. "(mor)-phonotactics" of the proto-language), and (3) replacement rules. Crucially, we want our tracing facility to provide not only apply-down traces of derivations from upper to lower forms, but also of apply-up derivations from lower to upper.

5.1 Special Problems

Special problems of programming the trace interpreter are posed by some reserved symbols. In particular, '**?**' for "any symbol" and '**0**' or '**[]**' and '**[..]**' for deletion and epenthesis rules, respectively, require attention. '**?**' is familiar enough and need not be discussed here.

The null-symbols '**0**' or '[]' and '[..]', however, can lead to difficulties with termination. For downward tracing there is no problem with deletion rules using '**0**' or '[]', whether they are conditioned by an environment or not, and tracing apply-down application of epenthesis rules with an

[1] Since both Perl and Prolog files have extension 'pl', the Perl scripts additionally have 'perl' in the filename before the 'pl' extension.

[2] All files that are created by the export module are saved to a temp directory and are deleted with the next execution of *ftrace.xfst*

environment is likewise unproblematic. An unconditioned epenthesis rule would be disastrous for a description and for tracing, but we assume one normally would not want to write such a rule in the first place for a natural language. This is all fairly obvious.

The situation is less transparent, however, when it comes to apply-up tracing. Again, there is no problem with deletion or epenthesis rules with environments, and – symmetrically to the apply-down application of deletion rules – even apply-up application of unconditioned epenthesis rules could in principle be handled, but we don't want such rules, anyway.

The real problem arises with unconditioned deletion rules. We have just seen that apply-down application is unproblematic, but their apply-up application is equivalent to apply-down application of unconditioned epenthesis, which we have excluded. So we appear to be faced with a dilemma: For many descriptions it appears attractive to have unconditioned rules to delete, e.g., symbols for morpheme boundaries, and in any case, we would not want to *disallow* their use by linguists; on the other hand, apply-up tracing seems inevitably to lead to an infinite or at least unacceptably large number of possible antecedent strings from which a given string could arise through uncondtioned deletion of a segment.

In order to deal with this we have developed a strategy based on the notion of distributional filtering. Consider the above-mentioned example of a rule '**'+' -> 0**' to delete all instances of a morpheme boundary '+' after it has served its function, e.g. in conditioning other morphophonemic rules. If our description consisted merely of a sequence of replacement rules *R1*, … , *Rn* including the deletion rule, then not only apply-up tracing with single rules, but also apply-up applications with the overall network in general would lead to an explosion in the computation of upper forms from which a lower form could arise. The problem dissolves, however, if we compose a network *N0* constraining the distribution of symbols in the ultimate upper language with the total network R arising from the composition of all replacement rules; if *N0* correctly specifies where '+' can occur in the first place, then it can only be deleted from these positions, and the problem is solved for apply up in a single, composite network.

We now need to carry over the filtering idea to upward tracing. The upper language defined by the network of a single unconditioned deletion rule must be restricted in order to ensure that the set of possible antecedent strings is highly constrained. Consider the sequence of networks *N0*, *N1*, …, *Nn* where each *Ni* except *N0* arises from *Ri*. For each *Nj* stemming from an unconditioned deletion rule, we can define *Nj'* as the composition *N0* .o. *N1* .o. … *Nj-1*. Then in upward tracing of the application of *Nj* to produce string *w*, not [*Nj* .o. *w*].u , but rather [*Nj'*.l .o. *Nj* .o. *w*].u is computed to get the set of possible antecedent strings. This gives us the desired filtering effect and solves the problem for tracing.

5.2 Downward tracing

The implementation of downward tracing is simple. Given replacement rules defined like these

```
define r1 [ k -> c || _ i ] ;
define r2 [ i -> 0 || _ .#. ] ;
```

xfst or foma constructs the network encoded in Prolog, which is exported to a file (cf. the example in Section 6 below).

Following the techniques of Gazdar and Mellish mentioned above, a Prolog network-interpreter for downward tracing can then be implemented easily. Due to limitations of space we omit the listings here. The interpreter has been tested extensively.

6 An Example

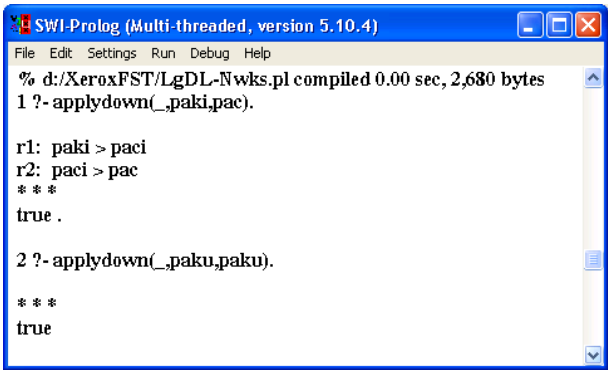
The following example already given above is very simple and transforms a fictitious proto-language *PL* into a daughter language *DL* with two ordered sound changes: palatalization of the velar *k* to *c* before the front vowel *i*, followed by the deletion of *i* in final position. Here, again, is the code of the language description:

```
# LgDL.txt
define r1 [ k -> c || _ i ] ;
define r2 [ i -> 0 || _ .#. ] ;
# define lgdl [r1 .o. r2] ; QQ
```

After the compilation of the language description, the Prolog networks of replacement rules **r1** and **r2** and the play list are exported to *LgDL-Nwks.pl*. Here is part of the content of this file:

```
:- encoding(utf8).
network(r1).
arc(r1, 0, 0, "?").
arc(r1, 0, 0, "c").
arc(r1, 0, 0, "i").
arc(r1, 0, 1, "k").
arc(r1, 0, 2, "k":"c").
arc(r1, 1, 0, "?").
arc(r1, 1, 0, "c").
arc(r1, 1, 1, "k").
arc(r1, 1, 2, "k":"c").
arc(r1, 2, 0, "i").
final(r1, 0).
final(r1, 1).
network(r2).
arc(r2, 0, 0, "?").
arc(r2, 0, 1, "i").
arc(r2, 0, 2, "i":"0").
arc(r2, 1, 0, "?").
arc(r2, 1, 1, "i").
arc(r2, 1, 2, "i":"0").
final(r2, 0).
final(r2, 2).
rule_list(lgdl, [r1, r2]).
```

The Prolog downward tracing interpreter *applydn.pl* is compiled in the SWI Prolog console, and then *LgDL-Nwks.pl* is consulted. Now the developer can test pairs of words from the proto-language and the daughter language:



7 Conclusion

We believe that FTrace can be useful and help developers of synchronic and diachronic language descriptions to debug their applications. In teaching historical linguistics it makes it possible to show the historical development of the phonological system of a language in detail and to test the proposed rules for derivations of individual forms. The same tool can equally well be used to produce explicit synchronic derivations from underlying forms to surface forms.

References

Kenneth R. Beesley and Lauri Karttunen. 2003. Finite State Morphology. CSLI, Stanford

Gerald Gazdar and Chris Mellish. 1989. Natural Language Processing in Prolog. Addison-Wesley, Wokingham et al.

Mans Hulden. 2009. Foma: a finite-state compiler and library. In: Proceedings of the EACL 2009 Demonstrations Session, pp. 29-32.

Lauri Karttunen. 1995. The replace operator. In: 33rd ACL Proceedings, 16-23.

Kemal Oflazer and Yasin Yılmaz. 2004. Vi-xfst: a visual regular expression development environment for Xerox finite state tool. In: SIGPHON 2004: Proceedings of the Seventh Meeting, Barcelona, Spain.