HFST Training Environment and Recent Additions

Axelson, Erik

Northern European Association for Language Technology
2023-04

Axelson , E , Hardwick , S & Linden , K 2023 , HFST Training Environment and Recent Additions . in Rule-Based Language Technology . NEALT Monograph Series , no. 2[1] , Northern European Association for Language Technology , Tartu , pp. 60-69 .

http://hdl.handle.net/10138/357039

HFST Training Environment and Recent Additions

Erik Axelson, Sam Hardwick, Krister Lindén
Department of Digital Humanities
FIN-00014 University of Helsinki, Finland
{erik.axelson,sam.hardwick,krister.linden}@helsinki.fi

Abstract

HFST - the Helsinki Finite-State Technology toolkit was launched in 2009 (Lindén & al, 2009) and has since been used for developing a number of rule-based morphologies for processing natural language. To promote the uptake of the toolkit a training environment for linguists to learn how to use HFST has been designed in Jupyter. This paper presents an overview of the training environment and some of the recent features that have been added to HFST to keep the run-time size of the transducer reasonably small despite exceptions and negative constraints that need to be added during practical FST development.

Key Words: *finite-state technology, morphological descriptions, negative constraints*

1 Introduction

In natural language processing, finite-state string transducer methods have been found useful for solving a number of practical problems ranging from language identification and optical character recognition via morphological processing and generation to part-of-speech tagging and named-entity recognition as well as machine translation, as long as the problems lend themselves to a formulation based on matching and transforming local context.

HFST–Helsinki Finite-State Technology (Lindén & al., 2009, 2011, 2013) is a framework for compiling and applying linguistic descriptions with finite-state methods. HFST connects some of the most important finite-state tools for creating morphologies and spellers into one open-source platform and supports extending and improving the descriptions with weights to accommodate the modelling of statistical information. HFST offers a path from language descriptions to efficient language applications in key environments and operating systems. HFST also provides an opportunity to exchange transducers between different software providers in order to get the best out of each finite-state library.

In this paper we present a training environment for linguists and some recent advances to help model builders optimise the size of their transducers for processing text from tokenization to recognizing semantic frames in context with HFST–Helsinki Finite-State Technology.

In Section 2, we present an overview of the interactive online training modules available in Jupyter. In Section 3, we present some advances in how to control the size of transducers that tend to grow rapidly in size when introducing exceptions to otherwise regular relations. In Section 4, we conclude the presentation.

2 Online Tutorials

As part of SAFMORIL (CLARIN Knowledge Centre for Systems and Frameworks for Morphologically Rich Languages, https://www.kielipankki.fi/safmoril/), we offer two online tutorials for XFST-based morphology development at the Notebooks service hosted by CSC (Finnish IT Center for Science): “Computational Morphology with HFST” and “Morphologically Rich Languages with HFST”.

The tutorials are implemented as Python notebooks which use the Python interface of HFST - Helsinki Finite-State Transducer Technology (Lindén & al, 2009, 2011, 2013). The tutorials are based on teaching materials from courses organized at the University of Helsinki. The tutorials mostly use the same examples and exercises as the original courses, but HFST command line tools have been replaced with HFST Python interface.

Prerequisites of the tutorials are foundations of general linguistics as well as basic knowledge on how to use a computer. Some programming experience is desirable, and knowledge of Natural Language Processing (NLP) is also a plus.

The tutorials contain several parts or “lectures”. A lecture usually contains some theory and simple examples where the user can test how things work in the HFST interface. At the end there are more difficult examples or “assignments” where the user has to find the solutions themselves and open questions to think about.

At the moment, access requires a HAKA Identity Federation account. If you do not have a HAKA account, you can contact SAFMORIL Helpdesk (safmoril@kielipankki.fi) to arrange for local accounts or request a visitor account directly from CSC service desk (servicedesk@csc.fi). You also need a join code that you can request from SAFMORIL Helpdesk. New logins are available during the fall (Sept 1 – Dec 15) and spring (Jan 15 – May 15) semesters.

2.1 Computational Morphology with HFST

The tutorial ”Computational Morphology with HFST” (Axelson et al., 2019) is an adaptation of the teaching material of the course ”Computational Morphology” taught by Mathias Creutz in the Master’s programme “Linguistic Diversity and Digital Humanities” at the University of Helsinki. The original course and its exercises have been created by Mathias Creutz and co-developed by Senka Drobac. Using an earlier version of the course material, Erik Axelson has developed a Jupyter notebook interactive version.

The tutorial is divided into seven parts:

1. Theories of morphology, generators and analysers, lexc
2. Finite-state basics, xfst rules
3. Disambiguation, probabilities, finite-state networks summarized
4. Guessers, stemmers, regular expressions in xfst
5. Twolc, two-level rules
6. Flag diacritics, non-concatenative morphology

7. Optimization of finite-state networks

2.1.1 Lecture 1

Theories of morphology and corresponding formalisms are introduced: word & paradigm (lexc, xfst), item & arrangement (lexc, xfst) and item & process (twolc, xfst). The student is familiarized with the concepts of generators and analysers. Lexc formalism is exemplified with a script that implements a simple morphological generator for English noun inflection.

2.1.2 Lecture 2

Finite-state basics and the underlying set theory (relations) for finite-state networks are introduced. The simple morphological generator for English noun inflection from the previous lecture is implemented and expanded with a different approach using a cascade of rule transducers defined with xfst formalism.

2.1.3 Lecture 3

The lecture starts with basics of probability theory and how probabilities can be used to disambiguate word forms and perform spelling correction. This leads to weighted transducers and rules that are introduced. In the end, a summary of types of finite-state automata and transducers is given.

2.1.4 Lecture 4

Guessers and stemmers are introduced. Esperanto verb guesser is used as a case study. Pronunciation lexica are defined for a language with (almost) regular orthography (Brazilian Portuguese) as well as for a language with highly irregular orthography (English). Regular expressions in xfst are introduced in more detail.

2.1.5 Lecture 5

Sequential xfst rules are revisited and parallel two-level twolc rules introduced. Morphology for English adjectives from lecture 2 is recalled and implemented with twolc. Twol rule operators are exemplified with consonant gradation in Finnish.

2.1.6 Lecture 6

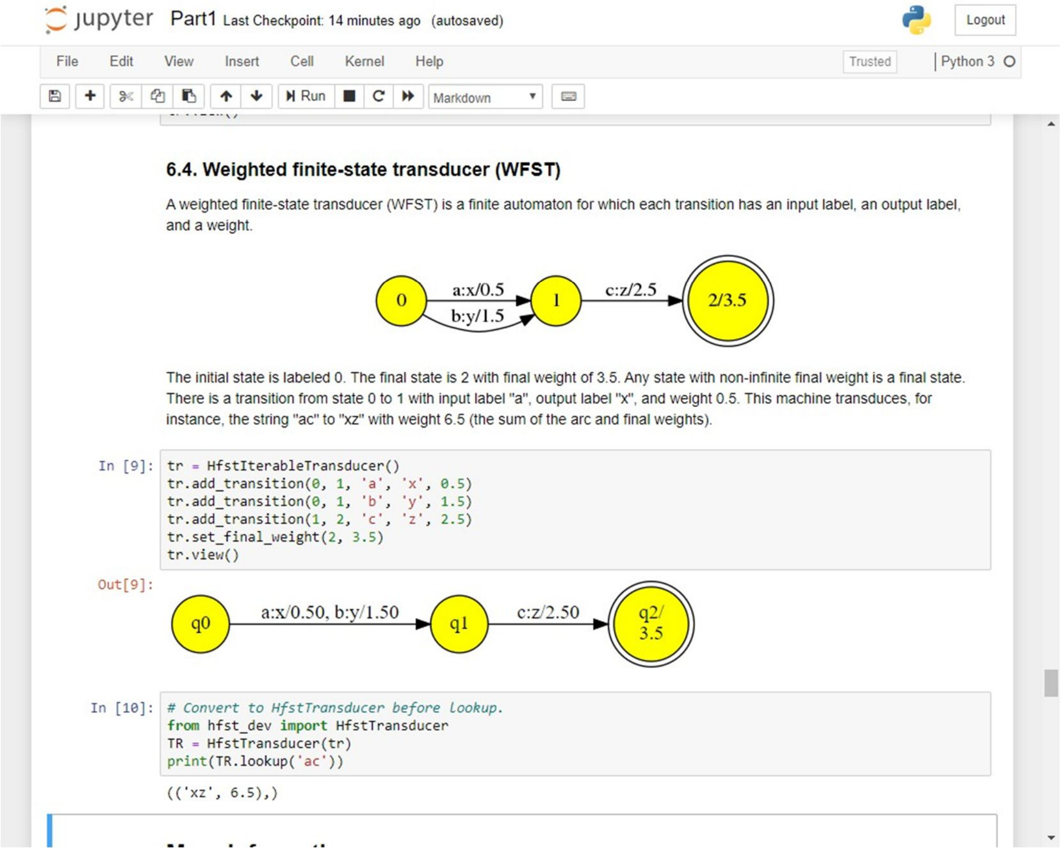
This lecture presents flag diacritics and their use in non-concatenative morphologies, such as Arabian, Tagalog and Malay. Different diacritic operators are listed with examples.

2.1.7 Lecture 7

The final lecture deals with optimization of finite-state networks. Determinization and minimization algorithms are introduced, both for unweighted and weighted networks.

For an example of a lecture module, see Figure 1.

Figure 1. A running instance of lecture in a Notebook session where weighted finite-state networks are exemplified and a simple network is constructed from scratch and a lookup is performed on it.



6.4. Weighted finite-state transducer (WFST)

A weighted finite-state transducer (WFST) is a finite automaton for which each transition has an input label, an output label, and a weight.

The initial state is labeled 0. The final state is 2 with final weight of 3.5. Any state with non-infinite final weight is a final state. There is a transition from state 0 to 1 with input label "a", output label "x", and weight 0.5. This machine transduces, for instance, the string "ac" to "xz" with weight 6.5 (the sum of the arc and final weights).

In [9]:
```
tr = HfstIterableTransducer()
tr.add_transition(0, 1, 'a', 'x', 0.5)
tr.add_transition(0, 1, 'b', 'y', 1.5)
tr.add_transition(1, 2, 'c', 'z', 2.5)
tr.set_final_weight(2, 3.5)
tr.view()
```

Out[9]:

In [10]:
```
# Convert to HfstTransducer before lookup.
from hfst_dev import HfstTransducer
TR = HfstTransducer(tr)
print(TR.lookup('ac'))
```

```
(('xz', 6.5),)
```

2.2 Morphologically Rich Languages with HFST

The tutorial “Morphologically Rich Languages with HFST” (Axelson et al., 2020) is based on a course by Jack Rueter & Sjur Moshagen organized at the University of Helsinki named “Language Technology for Finno-Ugric Languages - Methods, Tools and Applications”. The course belongs to the MA Programme “Linguistic Diversity in the Digital Age”. Using the course material, Erik Axelson and Jack Rueter have developed a Jupyter notebook interactive version.

This tutorial gives an introduction to mainly rule-based language technology as used in many full-scale, production projects using the GiellaLT and Apertium infrastructures. The technologies and methodologies presented can be used for any language, although the focus is on morphologically complex ones.

The course is divided into five parts. More parts may be added later.

1. Course intro, field overview, majority vs minority language technology
2. Intro to the Giella infra and a variety of finite-state tasks
3. Simple lexc (numerals, dates, clocks)
4. Writing test material and applying simple concatenative morphology
5. Twolc & xfst rewrite rules

2.2.1 Lecture 1

A short history of language technology (LT) is given. Reusability of grammars and LT tool components as well as scalability of infrastructure is also considered, in particular in the context of a minority language with few speakers.

2.2.2 Lecture 2

The Giella infra is introduced, emphasizing the templating system and the split between language-independent and language-specific code. The HFST interface is also introduced.

2.2.3 Lecture 3

Lexc formalism is exemplified with simple scripts where transducers for numerals, dates and clocks are created for Olonets Karelian. The differences and similarities between Olonets Karelian, Finnish and English are considered for numerals.

2.2.4 Lecture 4

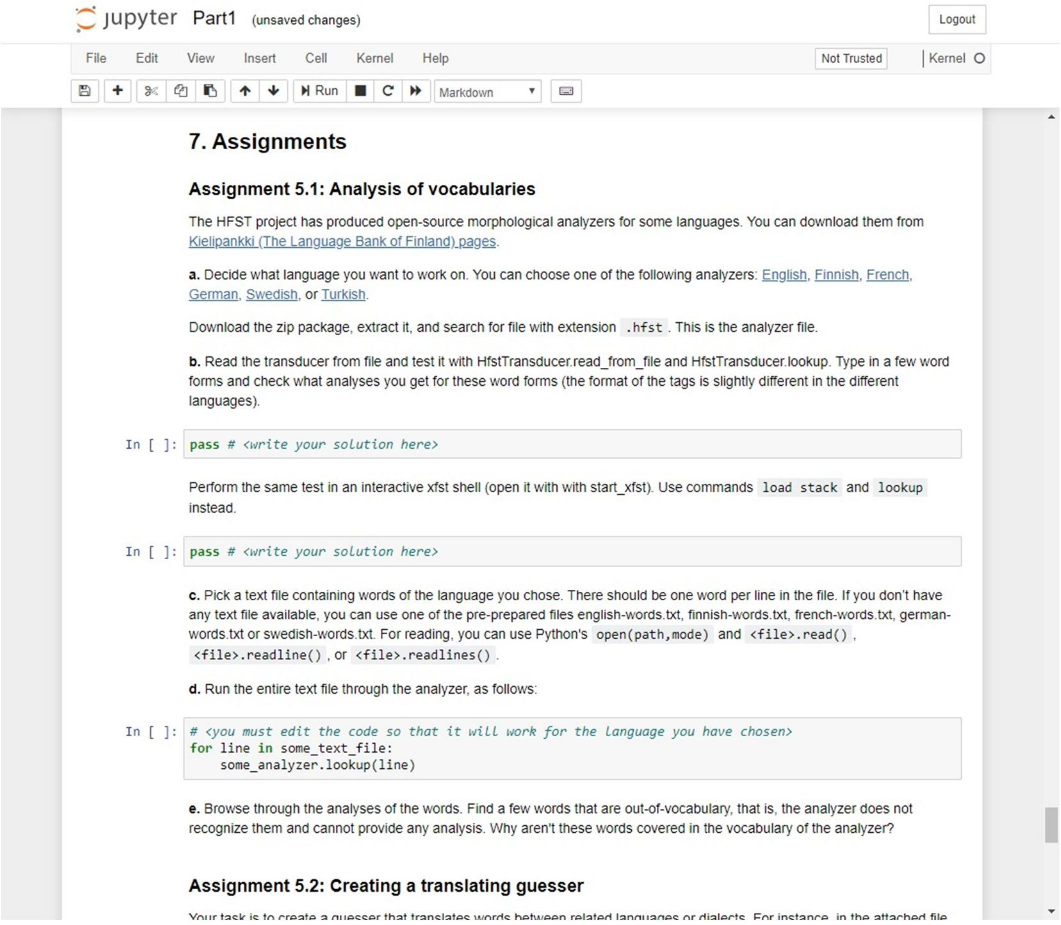
This lecture shows how test material should be written for a given morphology in the Giella infra. A simplified test case for Olonets Karelian nouns is created with a combination of concatenative morphology and phonological rules.

2.2.5 Lecture 5

Twolc formalism is introduced. An example from the previous lecture is revisited, using twolc rewrite rules. The differences between twolc and xfst rewrite rules are discussed.

For an example of a lecture module, see Figure 2.

Figure 2. A running instance of a lecture in a Notebook session where a morphological transducer is read from file, a few word forms are tested to see the analyses that the transducer gives to them, and finally a larger text file is given as input to the transducer to see how many out-of-vocabulary words are found. The user is also instructed to think of reasons for words that are not covered by the transducer.



7. Assignments

Assignment 5.1: Analysis of vocabularies

The HFST project has produced open-source morphological analyzers for some languages. You can download them from Kielipankki (The Language Bank of Finland) pages.

a. Decide what language you want to work on. You can choose one of the following analyzers: English, Finnish, French, German, Swedish, or Turkish.

Download the zip package, extract it, and search for file with extension `.hfst`. This is the analyzer file.

b. Read the transducer from file and test it with HfstTransducer.read_from_file and HfstTransducer.lookup. Type in a few word forms and check what analyses you get for these word forms (the format of the tags is slightly different in the different languages).

```
In [ ]: pass # <write your solution here>
```

Perform the same test in an interactive xfst shell (open it with with start_xfst). Use commands `load stack` and `lookup` instead.

```
In [ ]: pass # <write your solution here>
```

c. Pick a text file containing words of the language you chose. There should be one word per line in the file. If you don't have any text file available, you can use one of the pre-prepared files english-words.txt, finnish-words.txt, french-words.txt, german-words.txt or swedish-words.txt. For reading, you can use Python's `open(path,mode)` and `<file>.read()`, `<file>.readline()`, or `<file>.readlines()`.

d. Run the entire text file through the analyzer, as follows:

```
In [ ]: # <you must edit the code so that it will work for the language you have chosen>
        for line in some_text_file:
            some_analyzer.lookup(line)
```

e. Browse through the analyses of the words. Find a few words that are out-of-vocabulary, that is, the analyzer does not recognize them and cannot provide any analysis. Why aren't these words covered in the vocabulary of the analyzer?

Assignment 5.2: Creating a translating guesser

Your task is to create a guesser that translates words between related languages or dialects. For instance, in the attached file

2.3. Advanced features

Advanced features may be added to future versions of the tutorials, including the dynamic context constraints presented in Chapter 3.1. and arc compression presented in Chapter 3.2.

3 Relaxing the finite-state formalism

Finite-state methods have two major benefits. Firstly, they are predictable, in that their possible configurations are enumerated ahead of time. Secondly, they have an algebra, so they can be readily be operated on to obtain e.g. inverses and compositions with the same computational constraints. Their limitations include their locality: a finite-state machine

(FSM) can't combine information from multiple places in the input to make decisions, rather each possible combination must have its own state. If the relevant parts of input are separated by many intermediate steps, these combinations will include a large number of redundant states, leading to inefficiency.

In the history of finite-state methods, some linguistic features have been a sufficient problem that the methods have been extended. Non-consecutively appearing features with interactions may be kept track of with flag diacritics, a way to assign values to variables and use them to allow or disallow transitions to states. For example, a language may disallow a propositional prefix with certain case-marking suffixes. Without the flag diacritics, the entire repertoire of nouns must be repeated for the case where there was a prepositional suffix and the case without. With flag diacritics, as in Attia (2005), this can be avoided.

Certain applications have similarly revealed problems with the locality constraint. Rules based on named regular expressions are conceptually tractable and easy to write, but can explode in size, to the surprise of inexperienced rule-writers. We describe two methods of reducing these problems: one of which requires a change in rule-writing and formalism, and one which can be implemented on finished finite-state systems.

3.1 The problem

Consider a rule-based system for tagging named entities with appropriate labels. It will have a number of rules for proper names in general (probably using capitalization, knowledge about personal names, dates, addresses, etc.) environments in which certain named entities are likely to occur (in "the CEO of ______" the continuation is likely to be a corporation) as well as lists of known names of different types. In such a system for recognizing named entities in Finnish, FiNer (Ruokolainen et al. 2020) (1000+ rules and 100+ gazetteers), many situations called for rules of the type "here, allow anything, except for this list of exceptions". If something could be the name of a city or of a corporation, it is probably a city if it isn't one of a long list of known corporations.

In the regular expression syntax of XFST, one could write

```
?* - Exceptions
```

`?` is "any symbol" and the minus sign represents subtraction. This simple definition can blow up in terms of space when compiled. For example, if the exception is the word "exception", the FSM must allow any symbol, and provided that it read a "e", move to a state where it is ready to disallow the continuation "xception". Then, in that state, it has to be able to move to a state where the continuation to exclude is "ception", and so on.

For an implementation with a special symbol representing "any symbol not mentioned in the FSM" and regular symbols for all the other letters, this compiles to 11 states and 99 arcs for just one word. For when using the set of strings up to eight letters long known to a finite-state dictionary of Finnish, the expression compiles to 81 546 states and 27 807 186 arcs. In a real-world named entity recogniser ruleset in `hfst-pmatch` (Lindén et al., 2013) this problem duplicated over many parts of the final network resulted in a compiled size of over 700MB.

3.2 Dynamic subtraction

`hfst-pmatch`, as the Xerox `fst` (Karttunen 2011) from which it derives, features run-time context constraints. To require a rule to be followed by a particular right context, one can write `RC(context)` after the rule, where `context` is whatever context one wants to require. Similarly `NRC` for *negative* right context, and `LC` and `NLC` for left contexts. These contexts trigger special runtime behaviour, and are effectively an extension of the finite-state formalism, especially when combined with `pmatch`'s recursive transition networks (RTN). RTNs are stored separately from the rule, and can be used in multiple places without space overhead.

In `hfst-pmatch` the rule syntax was relaxed to allow right context constraints to appear not only on the right hand side of the rule, but anywhere. In particular, a right context can be on the left side of the rule. This effectively provides run-time subtraction:

```
NRC(Exception) ?*
```

Here, the FSM will try to match `Exception`, and proceed only if it fails to do so. The `NRC` directive compiles to two extra states and arcs.

3.3 Arc compression

Another more general phenomenon in large real-world finite-state applications is the tendency of FSMs to repeat common sets of symbols with the same target state in many places in the network. Sets of punctuation, digits, classes of alphabetic characters etc. are often allowed in many places in rules either explicitly and implicitly. If the rule-writer hasn't foreseen the problem, and used run-time features of the system (pre- or postprocessing, or character classes that can be represented by a single symbol), this can lead to bloated binaries. In the case of the previously mentioned NER system, millions of redundant arcs.

This is especially true of applications such as taggers and hyphenators, but can also apply to morphologies. For example, the Morphisto morphology for German contains thousands of occurrences of the digits 0-9 appearing together, with the same target.

If the intended runtime environment supports character set symbols, as `hfst-pmatch` does, the redundant arcs can be replaced as a post-compilation step. The `pmatch` syntax `Lst(Charset)` compiles to a special symbol which matches the symbols used in `Charset`.

3.4 Example outcome

In the previously mentioned named-entity recogniser, the original compiled size of over 700MB was reduced to 7MB using primarily dynamic subtraction. The source code for the original and optimized versions of the system may be found on the project's Github repository at https://github.com/Traubert/FiNer-rules. The release under tag 1.6.0 has the dynamic subtraction code in place.

4 Discussion and Conclusion

In this paper we presented a training environment in Jupyter for linguists. We also presented some recent advances to help linguists optimise the size of their transducers with HFST–Helsinki Finite-State Technology.

The HFTS tutorials are implemented as Python notebooks which use the Python interface of HFST. The tutorial ”Computational Morphology with HFST” (Axelson et al., 2019) is an adaptation of the teaching material of the course ”Computational Morphology” taught by Mathias Creutz teaching the basics of Finite-State Technology. The tutorial “Morphologically Rich Languages with HFST” (Axelson et al., 2020) is based on a course by Jack Rueter & Sjur Moshagen and is more directed towards practical solutions in linguistic descriptions.

As pointed out in the article, Finite-state methods have two major benefits. Firstly, they are predictable, in that their possible configurations are enumerated ahead of time. Secondly, they have an algebra, so they can be readily be operated on to obtain eg. inverses and compositions with the same computational constraints. However, one of their main drawbacks has been their locality constraint: a finite-state machine (FSM) can't easily combine information from multiple places in the input to make decisions, rather each possible combination must have its own state. In this article we described two methods for reducing these problems: one of which requires a change in rule-writing and formalism, and one which can be implemented on finished finite-state systems.

References

Attia, Mohammed A., 2005:
Developing a robust arabic morphological transducer using finite state technology. Paper presented at *The 8th Annual CLUK Research*, 2005.

Axelson Erik, Mathias Creutz, Senka Drobac, 2019:
Computational Morphology with HFST – an adaptation of the teaching material of the course “Computational Morphology” taught by Mathias Creutz in the Master’s programme “Linguistic Diversity and Digital Humanities” at the University of Helsinki. [Learning material]. Language bank of Finland. Accessible at http://urn.fi/urn:nbn:fi:lb-2021053001

Axelson Erik, Jack Rueter, Sjur Moshagen, 2020:
Morphologically Rich Languages with HFST – an adaptation of the teaching material of the course “Language Technology for Finno-Ugric Languages - Methods, Tools and Applications” taught by Jack Rueter and Sjur Moshagen in the Master’s programme “Linguistic Diversity in the Digital Age” at the University of Helsinki. [Learning material]. Language bank of Finland. Accessible at http://urn.fi/urn:nbn:fi:lb-2022082903

Karttunen Lauri, 2011:
Beyond Morphology: Pattern Matching with FST. In Cerstin Mahlow and Michael Piotrowski, editors, *Systems and Frameworks for Computational Morphology, volume 100 of Communications in Computer and Information Science*, pages 1–13, Berlin Heidelberg. Springer.

Lindén Krister, Erik Axelson, Senka Drobac, Sam Hardwick, Juha Kuokkala, Jyrki Niemi, Tommi Pirinen, Miikka Silfverberg, 2013:

HFST – a system for creating NLP tools. In: Mahlow, C., Piotrowski, M. (eds.) vol. 380, pp. 53–71. Springer, Berlin Heidelberg

Lindén Krister, Erik Axelson, Sam Hardwick, Tommi Pirinen, Miikka Silfverberg, 2011:
HFST—framework for compiling and applying morphologies. In: Mahlow, C., Piotrowski, M. (eds.) *Systems and Frameworks for Computational Morphology. Communications in Computer and Information Science*, vol. 100, pp. 67–85. Springer, Berlin Heidelberg.

Lindén Krister, Miikka Silfverberg, Tommi Pirinen, 2009:
HFST tools for morphology—an efficient opensource package for construction of morphological analyzers. In: Mahlow, C., Piotrowski, M. (eds.) *Systems and Frameworks for Computational Morphology. Lecture Notes in Computer Science*, vol. 41, pp. 28–47. Springer.

Ruokolainen Teemu, Pekka Kauppinen, Miikka Silfverberg, Krister Lindén, 2020:
A Finnish news corpus for named entity recognition. *Lang Resources & Evaluation 54*, 247–272.