Foma: a finite-state compiler and library

Mans Hulden
University of Arizona
mhulden@email.arizona.edu

Abstract

Foma is a compiler, programming language, and C library for constructing finite-state automata and transducers for various uses. It has specific support for many natural language processing applications such as producing morphological and phonological analyzers. Foma is largely compatible with the Xerox/PARC finite-state toolkit. It also embraces Unicode fully and supports various different formats for specifying regular expressions: the Xerox/PARC format, a Perl-like format, and a mathematical format that takes advantage of the 'Mathematical Operators' Unicode block.

1 Introduction

Foma is a finite-state compiler, programming language, and regular expression/finite-state library designed for multi-purpose use with explicit support for automata theoretic research, constructing lexical analyzers for programming languages, and building morphological/phonological analyzers, as well as spellchecking applications.

The compiler allows users to specify finite-state automata and transducers incrementally in a similar fashion to AT&T's fsm (Mohri et al., 1997) and Lextools (Sproat, 2003), the Xerox/PARC finite-state toolkit (Beesley and Karttunen, 2003) and the SFST toolkit (Schmid, 2005). One of Foma's design goals has been compatibility with the Xerox/PARC toolkit. Another goal has been to allow for the ability to work with n-tape automata and a formalism for expressing first-order logical constraints over regular languages and n-tape-transductions.

Foma is licensed under the GNU general public license: in keeping with traditions of free software, the distribution that includes the source code comes with a user manual and a library of examples.

The compiler and library are implemented in C and an API is available. The API is in many ways similar to the standard C library `<regex.h>`, and has similar calling conventions. However, all the low-level functions that operate directly on automata/transducers are also available (some 50+ functions), including regular expression primitives and extended functions as well as automata determinization and minimization algorithms. These may be useful for someone wanting to build a separate GUI or interface using just the existing low-level functions. The API also contains, mainly for spell-checking purposes, functionality for finding words that match most closely (but not exactly) a path in an automaton. This makes it straightforward to build spell-checkers from morphological transducers by simply extracting the range of the transduction and matching words approximately.

Unicode (UTF8) is fully supported and is in fact the only encoding accepted by Foma. It has been successfully compiled on Linux, Mac OS X, and Win32 operating systems, and is likely to be portable to other systems without much effort.

2 Basic Regular Expressions

Retaining backwards compatibility with Xerox/PARC and at the same time extending the formalism means that one is often able to construct finite-state networks in equivalent various ways, either through ASCII-based operators or through the Unicode-based extensions. For example, one can either say:

```
ContainsX = Σ* X Σ*;
MyWords = {cat}|{dog}|{mouse};
MyRule = n -> m || _ p;
ShortWords = [MyLex^1]_1 ∩ Σ^<6;
```

or:

Operators	**Compatibility variant**	**Function**
[ ] ()	[ ] ()	grouping parentheses, optionality
$\forall\ \exists$	N/A	quantifiers
$\backslash$ ‘		term negation, substitution/homomorphism
:	:	cross-product
$+\ *$	$+\ *$	Kleene closures
^<n ^>n ^{m,n}	^<n ^>n ^{m,n}	iterations
$_1\ _2$	.1 .2 .u .l	domain & range
.f	N/A	eliminate all unification flags
$\neg$ $ $. $?	~ $ $. $?	complement, containment operators
/ ./. /// $\backslash\backslash\backslash$ /\/	/ ./. N/A N/A	‘ignores’, left quotient, right quotient, ‘inside’ quotient
$\in\ \notin\ =\ \neq$	N/A	language membership, position equivalence
$\succ\ \prec$	< >	precedes, follows
$\vee\ \cup\ \wedge\ \cap$ - .P. .p.	| & − .P. .p.	union, intersection, set minus, priority unions
=> -> (->) @->	=> -> (->) @->	context restriction, replacement rules
$\|\|$	<>	shuffle (asynchronous product)
$\times\ \circ$	.x. .o.	cross-product, composition

Table 1: *The regular expressions available in Foma from highest to lower precedence. Horizontal lines separate precedence classes.*

```
define ContainsX ?* X ?*;
define MyWords {cat}|{dog}|{mouse};
define MyRule n -> m || _ p;
define ShortWords Mylex.i.l & ?^<6;
```

In addition to the basic regular expression operators shown in table 1, the formalism is extended in various ways. One such extension is the ability to use of a form of first-order logic to make existential statements over languages and transductions (Hulden, 2008). For instance, suppose we have defined an arbitrary regular language `L`, and want to further define a language that contains only one factor of `L`, we can do so by:

```
OneL = (∃x)(x ∈ L ∧ ¬(∃y)(y ∈ L
∧ ¬(x = y)));
```

Here, quantifiers apply to substrings, and we attribute the usual meaning to $\in$ and $\wedge$, and a kind of concatenative meaning to the predicate $S(t_1, t_2)$. Hence, in the above example, `OneL` defines the language where there exists a string x such that x is a member of the language L and there does not exist a string y, also in L, such that y would occur in a different position than x. This kind of logical specification of regular languages can be very useful for building some languages that would be quite cumbersome to express with other regular expression operators. In fact, many of the internally complex operations of Foma are built through a reduction to this type of logical expressions.

3 Building morphological analyzers

As mentioned, Foma supports reading and writing of the LEXC file format, where morphological categories are divided into so-called continuation classes. This practice stems back from the earliest two-level compilers (Karttunen et al., 1987). Below is a simple example of the format:

```
Multichar_Symbols +Pl +Sing
LEXICON Root
        Nouns;

LEXICON Nouns
cat     Plural;
church  Plural;

LEXICON Plural
+Pl:%^s       #;
+Sing         #;
```

4 An API example

The Foma API gives access to basic functions, such as constructing a finite-state machine from a regular expression provided as a string, performing a transduction, and exhaustively matching against a given string starting from every position.

The following basic snippet illustrates how to use the C API instead of the main interface of Foma to construct a finite-state machine encoding the language a^+b^+ and check whether a string matches it:

```
1. void check_word(char *s) {
2.   fsm_t *network;
3.   fsm_match_result *result;
4.
5.   network = fsm_regex("a+ b+");
6.   result = fsm_match(fsm, s);
7.   if (result->num_matches > 0)
8.     printf("Regex matches");
9.
10 }
```

Here, instead of calling the fsm_regex() function to construct the machine from a regular expressions, we could instead have accessed the beforementioned low-level routines and built the network entirely without regular expressions by combining low-level primitives, as follows, replacing line 5 in the above:

```
network = fsm_concat(
          fsm_kleene_plus(
          fsm_symbol("a")),
          fsm_kleene_plus(
          fsm_symbol("b")));
```

The API is currently under active development and future functionality is likely to include conversion of networks to 8-bit letter transducers/automata for maximum speed in regular expression matching and transduction.

5 Automata visualization and educational use

Foma has support for visualization of the machines it builds through the AT&T Graphviz library. For educational purposes and to illustrate automata construction methods, there is some support for changing the behavior of the algorithms.

For instance, by default, for efficiency reasons, Foma determinizes and minimizes automata between nearly every incremental operation. Operations such as unions of automata are also constructed by default with the product construction method that directly produces deterministic automata. However, this on-the-fly minimization and determinization can be relaxed, and a Thompson construction method chosen in the interface so that automata remain non-deterministic and non-minimized whenever possible—non-deterministic automata naturally being easier to inspect and analyze.

6 Efficiency

Though the main concern with Foma has not been that of efficiency, but of compatibility and extendibility, from a usefulness perspective it is important to avoid bottlenecks in the underlying algorithms that can cause compilation times to skyrocket, especially when constructing and combining large lexical transducers. With this in mind, some care has been taken to attempt to optimize the underlying primitive algorithms. Table 2 shows a comparison with some existing toolkits that build deterministic, minimized automata/transducers. One the whole, Foma seems to perform particularly well with pathological cases that involve exponential growth in the number of states when determinizing non-deterministic machines. For general usage patterns, this advantage is not quite as dramatic, and for average use Foma seems to perform comparably with e.g. the Xerox/PARC toolkit, perhaps with the exception of certain types of very large lexicon descriptions (>100,000 words).

	Foma	xfst	GNU flex	AT&T fsm 4
$\Sigma^* a \Sigma^{15}$	0.216s	16.23s	17.17s	1.884s
$\Sigma^* a \Sigma^{20}$	8.605s	nf	nf	153.7s
North_Sami	14.23s	4.264s	N/A	N/A
8queens	0.188s	1.200s	N/A	N/A
sudoku2x3	5.040s	5.232s	N/A	N/A
lexicon.lex	1.224s	1.428s	N/A	N/A
3sat30	0.572s	0.648s	N/A	N/A

Table 2: *A relative comparison of running a selection of regular expressions and scripts against other finite-state toolkits. The first and second entries are short regular expressions that exhibit exponential behavior. The second results in a FSM with 2^{21} states and 2^{22} arcs. The others are scripts that can be run on both Xerox/PARC and Foma. The file lexicon.lex is a LEXC format English dictionary with 38418 entries. North_Sami is a large lexicon (lexc file) for the North Sami language available from* `http://divvun.no`.

7 Conclusion

The Foma project is multipurpose multi-mode finite-state compiler geared toward practical construction of large-scale finite-state machines such as may be needed in natural language processing as well as providing a framework for research in finite-state automata. Several wide-coverage morphological analyzers specified in the LEXC/xfst format have been compiled successfully with Foma. Foma is free software and will remain under the GNU General Public License. As the source code is available, collaboration is encouraged.

References

Beesley, K. and Karttunen, L. (2003). *Finite-State Morphology*. CSLI, Stanford.

Hulden, M. (2008). Regular expressions and predicate logic in finite-state language processing. In Piskorski, J., Watson, B., and Yli-Jyrä, A., editors, *Proceedings of FSMNLP 2008*.

Karttunen, L., Koskenniemi, K., and Kaplan, R. M. (1987). A compiler for two-level phonological rules. In Dalrymple, M., Kaplan, R., Karttunen, L., Koskenniemi, K., Shaio, S., and Wescoat, M., editors, *Tools for Morphological Analysis*. CSLI, Palo Alto, CA.

Mohri, M., Pereira, F., Riley, M., and Allauzen, C. (1997). AT&T FSM Library-Finite State Machine Library. *AT&T Labs—Research*.

Schmid, H. (2005). A programming language for finite-state transducers. In Yli-Jyrä, A., Karttunen, L., and Karhumäki, J., editors, *Finite-State Methods and Natural Language Processing FSMNLP 2005*.

Sproat, R. (2003). Lextools: a toolkit for finite-state linguistic analysis. *AT&T Labs—Research*.