Finite-State Technology as a Programming Environment

Shuly Wintner

Department of Computer Science, University of Haifa, 31905 Haifa, Israel
shuly@cs.haifa.ac.il

Abstract. Finite-state technology is considered the preferred model for representing the phonology and morphology of natural languages. The attractiveness of this technology for natural language processing stems from four sources: modularity of the design, due to the closure properties of regular languages and relations; the compact representation that is achieved through minimization; efficiency, which is a result of linear recognition time with finite-state devices; and reversibility, resulting from the declarative nature of such devices.

However, when wide-coverage grammars are considered, finite-state technology does not scale up well, and the benefits of this technology can be overshadowed by the limitations it imposes as a programming environment for language processing. This paper focuses on several aspects of large-scale grammar development. Using a real-world benchmark, we compare a finite-state implementation with an equivalent Java program with respect to ease of development, modularity, maintainability of the code and space and time efficiency. We identify two main problems, *abstraction* and *incremental development*, which are currently not addressed sufficiently well by finite-state technology, and which we believe should be the focus of future research and development.

1 Introduction

Finite-state technology (FST) denotes the use of finite-state devices, such as automata and transducers, in natural language processing (NLP). Since the early works which demonstrated the applicability of this technology to linguistic representation [1,2,3], FST is considered adequate for describing the phonological and morphological processes of the world's languages [4,5]. Even non-concatenative processes such as circumfixation, root-and-pattern morphology or reduplication, were shown to be in principle implementable in FST [6,7].

The utility of FST for NLP was emphasized by the implementation of several toolboxes which provide extended regular expression languages and compilers which convert expressions to finite-state automata and transducers. These include *INTEX* [8]; *FSM* [9], which is a unix-based set of programs for manipulating automata and transducers; *FSA Utilities* [10], which is a freely available, Prolog implemented system; and *XFST* [5], which is a commercial package assumed to be the most suitable for linguistic applications by providing the most expressive language.

A. Gelbukh (Ed.): CICLing 2007, LNCS 4394, pp. 97–106, 2007.

The benefits of FST for NLP stem from several properties of finite-state devices:

True representation: Following the pioneering work of Johnson [1], it is now clear that the kind of phonological and morphological rules that are common in linguistic theories can be directly implemented as finite-state relations. The implementation of linguistically motivated rules in FST is therefore straightforward and direct [11].

Modularity: The closure properties of regular languages and relations provide various means for combining regular expressions, supporting a variety of operations on the languages these expressions denote. For example, closure under *union* facilitates a separate development of two grammar fragments which can then be directly combined in a single operation. The most useful operations under which transductions are closed is probably *composition*, which is the central vehicle for implementing *replace rules* [3,11].

Compactness: Finite-state automata can be minimized, guaranteeing that for a given language, an automaton with a minimal number of states can always be generated. Toolboxes can apply minimization either explicitly or implicitly to improve storage requirements.

Efficiency: When an automaton is deterministic, recognition is optimally efficient (linear in the length of the string to be recognized). Automata can always be determinized, and toolboxes can take advantage of this to improve time efficiency.

Reversibility: Finite-state automata and transducers are inherently declarative: it is the application program which either implements recognition or generation. In particular, transducers can be used to map strings from the upper language to the lower language or vice versa with no changes in the underlying finite-state device.

These benefits encouraged the development of several large-scale morphological grammars for a variety of languages, including some with complex morphology such as German, French, Finnish, Turkish, Arabic and Hebrew.

The main claim of this paper, however, is that finite-state technology is still inferior to its alternatives when the development of large-scale grammars is concerned. This claim is supported by a realistic experiment defining a sophisticated morphological task, both using FST (section 2) and with a direct implementation in Java of the same grammar (section 3). We compare the two approaches in section 4 along several axes. The conclusion (section 5) is the identification of two main Achilles Heels in contemporary technology: the lack of *abstraction* mechanisms and the computational burden of incremental changes. We believe that these two issues should be the focus of future research in finite-state technology.

2 A Motivating Example

In order to evaluate the scalability of finite-state technology we consider, as a benchmark, a large-scale task: accounting for the morphological and ortho-

graphic phenomena of Hebrew, a natural language with non-trivial morphology. Clearly, languages with simple morphology (e.g., English) do not benefit from FST approaches, simply because it is so inexpensive to generate and store all the inflected forms. It is only when relatively complicated morphological processes are involved that the benefits of FST become apparent, and Hebrew is chosen here only as a particular example; the observations reported in section 4 are valid in general, for all similar tasks.

Hebrew, like other Semitic languages, has a rich and complex morphology. The major word formation machinery is root-and-pattern, where roots are sequences of three (typically) or more consonants and patterns are sequences of vowels and, sometimes, also consonants, with "slots" into which the root's consonants are inserted. After the root combines with the pattern, some morpho-phonological alterations take place, which may be non-trivial. The combination of a root with a pattern produces a *lexeme*, which can then be inflected in various forms. Inflectional morphology is highly productive and consists mostly of suffixes, but sometimes of prefixes or circumfixes. The morphological problems are amplified by issues of orthography. The standard Hebrew script leaves most of the vowels unspecified. Furthermore, many particles, including prepositions, conjunctions and the definite article, attach to the words which immediately follow them. As a result, surface forms are highly ambiguous.

The finite-state grammar which we used as a benchmark here is HAMSAH [12], an XFST implementation of Hebrew morphology. The grammar is obtained by composing a large-scale lexicon of Hebrew (over 20,000 entries) with a large set of rules, implementing mostly morphological and orthographic processes in the language. As the lexicon is developed independently [13] and is represented in XML, it must be converted to XFST before it can be incorporated in the grammar. This is done by a set of Perl scripts which had to be specifically written for this purpose. In other words, the system itself is not purely finite-state, and we maintain that few large-scale systems for morphological analysis can be purely finite-state, as such systems must interact with independently developed components such as lexicons, annotation tools, user interfaces etc.

A specialized set of rules implements the morphological processes which apply to each major part of speech. For example, figure 1 depicts a somewhat simplified version of the rule which accounts for the *wt* suffix of Hebrew nouns. This rule makes extensive use of composition (denoted by '`.o.`') and replace rules ('`->`' and '`<-`'). The effect of this rule is dual: on the surface level, it accounts for alterations in the concatenation of the suffix with the stem (e.g., *iih* becomes *ih*, *wt* changes to *wi* and a final *h* or *t* are elided); on the lexical level, it changes the specification of *number* from *singular* to *plural*.

The rule should be read from the center outwards. The variable `noun` denotes the set of all lexical items whose part of speech is noun; by default, these nouns are singular masculine. In XFST, a set of words is identified with with the identity transduction which relates each word in the set with itself. The first composition on top of the `noun` transduction selects only those nouns whose `plural` attribute is lexically specified as `wt` (other nouns may be lexically speci-

fied for a different plural suffix). Of those, only the ones whose `number` attribute is `singular` are selected. Then, the value `singular` in the lexical (upper) string is replaced by `plural` in the context of immediately following the attribute `number`. In the surface (lower) language, meanwhile, a set of compositions operators takes care of the necessary orthographic changes, and finally, the plural suffix *wt* is concatenated to the end of the surface string.

```
define pluralWTNoun [
 [
  [ plural <- singular || number _ ]
  .o. $[number singular]
  .o. $[plural wt]
  .o. noun
  .o. [ i i h -> i h || _ .#. ]
  .o. [ i t -> i || _ .#. ]
  .o. [ w t -> w i || _ .#. ]
  .o. [ [h|t] -> 0 || _ .#. ]
 ] [ 0 .x. [w t] ]
];
```

Fig. 1. XFST account of plural nouns

This rule is a good example of how a single phenomenon is factored out and accounted for independently of other phenomena: the rule refers to lexical information, such as '`number`' or '`plural`', but completely ignores irrelevant information such as, say, gender. However, it also hints at how information is manipulated by regular expressions. Since finite-state networks have no memory, save for the state, all information is encoded by strings which are manipulated by the rules. Thus, a simple operation such as changing the value of the *number* feature from *singular* to *plural* must be carried out by the same replace rules which account for the changes to the surface form. Furthermore, there is no way to structure such information, as is common in programming languages; and there is no way to encapsulate it.

3 An Alternative Implementation

We re-implemented the HAMSAH grammar directly as a Java program. The method we used was *analysis by generation*: we first generated all the inflected forms induced by the lexicon and store them in a database; then, analysis is simply a database lookup. It is common to think that for languages with rich morphology such a method is impractical. While this may have been the case in the past, contemporary computers can efficiently store and retrieve millions of inflected forms. Of course, this method would break in the face of an infinite lexicon (which can easily be represented with FST), but for most practical purposes it is safe to assume that natural language lexicons are finite.

To separate linguistic knowledge from processing code as much as possible, our Java implementation uses a database of *rules*, which are simple string transductions intended to account for simple (mostly morpheme boundary) morphological and orthographic alterations. When generating inflected forms, the program identifies certain conditions (e.g., a plural suffix *wt* is to be attached to a noun). It then looks up this condition in the rule database and retrieves the action to apply, depending on the suffix of the input string. An example of the rule database, with alterations pertaining to the suffix *wt* (cf. figure 1), is depicted in figure 2. For most morphological processes, solutions such as this can accurately stand for linguistic rules of the form depicted in figure 1.

When input ends in:	*iih*	*it*	*wt*	*h,*	*t*	default
Replace it by:	*ih*	*i*	*wi*	ϵ		
Then add:	*wt*	*wt*	*wt*	*wt*	*wt*	

Fig. 2. Direct account of plural nouns

Note that rules such as the one depicted in figure 2 are *generation* rules, and must not be confused with the kind of ad-hoc rules used at run time for, e.g., stemming. They fully reflect the linguistic knowledge encoded in finite-state replace rules. Granted, the example rule is simplistic, and more complex phenomena require more complicated representation, but since most of morphology takes place along morpheme boundaries, this is a reasonable representation.

The morphological analyzer was obtained by directly implementing the rules and applying them to the lexicon. The number of inflected forms (before attaching prefixes) is 473,880 (over 300,000 of those are inflected nouns, and close to 150,000 are inflected verb forms). In addition to inflected forms, the analyzer also allows as many as 174 different sequences of prefix particles to be attached to words; separation of prefixes from inflected forms is done at analysis time. The direct implementation is equivalent to the finite-state grammar: this was verified by exhaustively generating all the inflected forms with each of the systems and analyzing them with the other system.

4 Comparison and Evaluation

Having described the XFST benchmark grammar and its direct Java implementation, we now compare the two approaches along several axes. It is important to emphasize that we do not wish to compare the two systems, but rather the methodology. In particular, we chose XFST as it is one of the most efficient, and certainly the most expressive, FST toolbox available. A recent comparison of XFST with the FSA Utilities package [14] shows that the latter simply cannot handle grammars of the scale of HAMSAH. The following is a list of issues in which finite-state technology turned out to be problematic compared with the alternative; in the next section we focus on issues that we believe should be given

more attention in future research on FST. All experiments were done on a dual 2GHz processor Linux machine with 2.5Gb of memory.

Truthfulness. One of the assets of FST is that it allows for a very accurate implementation of linguistic rules. However, a good organization of the software can provide a clear separation between linguistic knowledge and processing in any programming environment, so that linguistic rules can be expressed concisely and declaratively, as exemplified in figure 2.

Reversibility. A clear advantage of FST is that grammars are fully reversible. However, with the analysis by generation paradigm the same holds also for a direct implementation: the generator is directly implemented, and the analyzer is implemented as search in the database of generated forms.

Expressivity. Here, the disadvantages of finite-state technology as a programming environment are clear. Programming with finite-state technology is very different from programming in ordinary languages, mainly due to the highly constrained expressive power of regular relations (programmers sometimes feel that they are working with their hands tied behind their backs). While FST can theoretically account for non-concatenative processes, existing toolboxes provide a partial, and sometimes overly complicated, solutions for such problems. Sometimes a trans-regular operation is called for, and many other times the constrained expressivity of regular relations is too limiting.

Portability. XFST is a proprietary package with three versions available for three common operating systems. Other finite-state toolboxes are freer; FSA is open source, but as we noted earlier it simply cannot cope with grammars the size of HAMSAH. FSM is available for a variety of Unix operating systems, as a binary only, whereas INTEX is distributed as a Windows executable. In contrast, a Java implementation can be delivered to users with all kinds of (contemporary) operating systems and hardware, and is optimally portable. The practical portability limitations directly hamper the utilization of finite-state technology in practical, commercial systems.

Abstraction. Large-scale morphological grammars tend to be extremely *non*-modular. Each surface string is associated, during its processing, with a lexical counterpart which describes its structure. The lexical string is constantly rewritten by the rules, as in figure 1. Due to the inherent sequentiality of strings, all the information which is associated with surface strings is encoded sequentially. In particular, adding a piece of information (e.g., adding the feature *gender* to an existing grammar which did not specify this property) requires a change in all the rules which account for this information; there is no way to abstract away from the actual implementation of this information, and the grammar developer must be consistent with respect to where this information is specified (i.e., whether it precedes or follows information on *number*).

Since information cannot be encapsulated and the language provides no abstraction mechanisms, collaborative development of finite-state grammars is difficult. All grammar developers must be aware of how information is represented

at all times. Furthermore, since the only data type is strings, debugging becomes problematic: very few errors can be detected at compile time.

In contrast, a direct Java implementation benefits from all the advantages of developing in an object-oriented environment. For example, the modules which inflect nouns and adjectives inherit from the same module, accounting for all *nominals*, which in turn inherits from a general module of inflection rules.

Collaborative development. A different facet of modularity has to do with the qualifications of the grammar developers. In order to take advantage of the full power of XFST, grammar developers must be simultaneously trained linguists and experienced programmers. With a direct implementation, a true interdisciplinary collaboration is enabled where a linguist can be in charge of characterizing the linguistic phenomena (and building the rule database) and a programmer can be responsible only for the actual implementation.

Maintenance. A by-product of the non-modularity of FST grammars is that maintaining them is difficult and expensive. It is hard to find a single person who is knowledgeable in all aspects of the design, and any change in the grammar is painful. This must be added to the poor compile-time performance, which again hampers maintainability.

Compile-time efficiency. A major obstacle in the development of XFST grammars is the speed of compilation. As is well known, many of the finite-state operators result in huge networks: theoretically, composition of networks of m and n states yields a network with $O(m \times n)$ states, and replace rules are implemented using composition. This leads to temporary networks which are sometimes larger than the available memory, requiring disk access and thereby slowing compilation down dramatically. While automata can always be minimized, this is not the case for transducers [15].

Theoretically, it is very easy to come up with very small regular expressions whose compilation is intractable. For any integer $n > 2$, there exists an n-state automaton A, such that any automaton that accepts the complement of $L(A)$ needs at least 2^{n-2} states [16]. An example of an XFST expression whose compilation time is exponential in n is: `~[ [a|b]* a [a|b]^n b [a|b]* ]`. In practice, the complete Hebrew grammar is represented, in XFST, by a network of approximately 2 million states and 2.2 million transitions. Compiling the entire network takes over 48 minutes and requires 3Gb of memory.

Compilation time is usually considered a negligible criterion for evaluating system performance. However, when developing a large-scale system, the ability to make minor changes and quickly re-make the system is crucial. With XFST, modification of even a single lexical entry requires at least an intersection of (the XFST representation of) this entry with the network representing the rules which apply to it. As a concrete example, adding a single two-character proper name (which does not inflect) to the lexicon increased the size of the network by 9 states and 10 arcs, but took almost three minutes to compile. Adding a two-character adjective resulted in the addition of 27 states and 30 arcs, and took about the same time.

In the direct implementation, modification of a single lexical entry requires generation of all inflected forms of this entry, which takes a fraction of a second; the time it takes to generate k lexical entries is proportional to k and is independent of the size of the remainder of the system. The analysis program is not altered.

To summarize the differences, figure 3 shows the time it takes to compile a network when k lexical entries are modified, for three values of k, corresponding to the number of adjectives, adverbs and the size of the entire lexicon. This time is compared with the time it takes to generate all inflected forms of these sub-lexicons in the analysis by generation paradigm.

#items	360 (adverbs)	1,648 (adjectives)	21,400 (all)
FST	13:47	13:55	48:12
Java	0:14	3:59	30:34

Fig. 3. Compilation/generation times (in minutes) when some lexical items change

Run-time efficiency. While finite-state automata guarantee linear recognition time, this is not the case with transducers, which cannot always be determinized [17]. Even when a device can be determinized, the determinization algorithm is inefficient (theoretically, the size of the deterministic automaton can be exponential in the size of its non-deterministic counterpart).

As it turns out, storing a database of half a million inflected forms (along with their analyses) is inexpensive, and retrieving items from the database can be done very efficiently. We experimented with two versions: one uses MySQL as the database and the other loads the inflected forms into a hash table. In this latter version, most of the time is spent on loading the database, and retrieval time is negligible.

We compared the performance of the two systems on four tasks, analyzing text files of 10, 100, 1,000 and 10,000 tokens. The results are summarized in figure 4, and clearly demonstrate the superiority of the direct implementation. In terms of memory requirements, XFST requires approximately 57Mb of memory, whereas the Java implementation uses no more than 10Mb. This is not a significant issue with contemporary hardware.

5 Discussion

We compared the process of developing a large-scale morphological grammar for Hebrew with finite-state technology with a direct implementation of the morphological rules in Java. Our conclusion is that finite-state technology remains superior to its alternatives with respect to the true representation of linguistic knowledge, and is therefore more adequate for smaller-scale grammars, especially those whose goal is to demonstrate specific linguistic phenomena rather than form the basis of large practical systems. However, viewed as a programming

#Tokens	10	100	1,000	10,000
FST	1.25	2.40	12.97	118.71
Java+MySQL	1.24	3.04	8.84	44.94
Java+Hash	5.00	5.15	5.59	7.64

Fig. 4. Time performance of both analyzers (in seconds)

environment, FST suffers from severe limitations, the most significant of which are lack of abstraction and difficulties in incremental processing.

Abstraction is the essence of computer science and the key to software development. Working with regular expressions and developing rules which use strings as the only data structure does not leave much space for sophisticated abstraction. Several works attempt to remedy this problem. XFST itself provides a limited solution, in the form of *flag diacritics* [5]. These are feature-value pairs which can be added to the underlying machines in order to add limited memory to networks; a similar solution, which is fully worked-out mathematically, is provided by *finite-state registered automata* [7]. These approaches are too low-level to provide the kind of abstraction that programmers have become used to. A step in the right direction is the incorporation of feature structures and unification into finite-state transducers [18], and in particular the recent proposal to use typed feature structures as the entities on which such transducers operate [19]. More research is needed in order to fully develop this direction and incorporate its consequences into a finite-state based grammar development framework.

The problem of incremental grammar development, exemplified in figure 3, can also be remedied by incorporating some recent theoretical results, in particular in incremental construction of lexicons [20,21], into an existing framework. Ordinary programming languages benefit from decades of research and innovation in compilation theory and optimization. In order for finite-state technology to become a viable programming environment for natural language morphology applications, more research is needed along the lines suggested here.

Acknowledgments. This work was funded by the Israeli Ministry of Science and Technology, under the auspices of the Knowledge Center for Processing Hebrew. The research was supported by a grant from the Israel Internet Association. I am very grateful to Shlomo Yona for implementing the XFST grammar and to Dalia Bojan for implementing the Java system. Kemal Oflazer provided useful comments on an earlier version of this paper. I also wish to thank Yael Cohen-Sygal, Alon Itai, Nurit Melnik and Shira Schwartz for their help. The views expressed in this paper as well as all remaining errors are, of course, my own.

References

1. Johnson, C.D.: Formal Aspects of Phonological Description. Mouton, The Hague (1972)
2. Koskenniemi, K.: Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production. The Department of General Linguistics, University of Helsinki (1983)

3. Kaplan, R.M., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20** (1994) 331–378
4. Roche, E., Schabes, Y., eds.: Finite-State Language Processing. Language, Speech and Communication. MIT Press, Cambridge, MA (1997)
5. Beesley, K.R., Karttunen, L.: Finite-State Morphology: Xerox Tools and Techniques. CSLI, Stanford (2003)
6. Beesley, K.R.: Arabic morphology using only finite-state operations. In Rosner, M., ed.: Proceedings of the Workshop on Computational Approaches to Semitic languages, Montreal, Quebec, COLING-ACL'98 (1998) 50–57
7. Cohen-Sygal, Y., Wintner, S.: Finite-state registered automata for non-concatenative morphology. Computational Linguistics **32** (2006) 49–82
8. Silberztein, M.: Dictionnaires électroniques et analyse automatique de textes : le système INTEX. Masson, Paris (1993)
9. Mohri, M., Pereira, F., Riley, M.: The design principles of a weighted finite-state transducer library. Theoretical Computer Science **231** (2000) 17–32
10. van Noord, G., Gerdemann, D.: An extendible regular expression compiler for finite-state approaches in natural language processing. In Boldt, O., Jürgensen, H., eds.: Automata Implementation. Number 2214 in Lecture Notes in Computer Science. Springer (2001)
11. Karttunen, L.: The replace operator. In: Proceedings of the Annual Meeting of the Association for Computational Linguistics. (1995) 16–23
12. Yona, S., Wintner, S.: A finite-state morphological grammar of Hebrew. Natural Language Engineering (Forthcoming)
13. Itai, A., Wintner, S., Yona, S.: A computational lexicon of contemporary Hebrew. In: Proceedings of The fifth international conference on Language Resources and Evaluation (LREC-2006), Genoa, Italy (2006)
14. Cohen-Sygal, Y., Wintner, S.: XFST2FSA: Comparing two finite-state toolboxes. In: Proceedings of the ACL-2005 Workshop on Software, Ann Arbor, MI (2005)
15. Mohri, M.: Minimization algorithms for sequential transducers. Theoretical Computer Science **234** (2000) 177–201
16. Holzer, M., Kutrib, M.: State complexity of basic operations on nondeterministic finite automata. In: Implementation and Application of Automata (CIAA '02). (2002) 151–160
17. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics **23** (1997) 269–312
18. Zajac, R.: Feature structures, unification and finite-state transducers. In: FSMNLP'98: The International Workshop on Finite-state Methods in Natural Language Processing, Ankara, Turkey (1998)
19. Amtrup, J.W.: Morphology in machine translation systems: Efficient integration of finite state transducers and feature structure descriptions. Machine Translation **18** (2003) 217–238
20. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite-state automata. Computational Linguistics **26** (2000) 3–16
21. Carrasco, R.C., Forcada, M.L.: Incremental construction and maintenance of minimal finite-state automata. Computational Linguistics **28** (2002) 207–216