Arabic Morphology Parsing Revisited

Suhel Jaber and Rodolfo Delmonte

University Ca' Foscari, Dept. Language Sciences, Laboratory Computational Linguistics,
Ca' Bembo, Dorsoduro 1705, 30123 Venezia, Italy
{jaber,delmont}@unive.it

Abstract. In this paper we propose a new approach to the description of Arabic morphology using 2-tape finite state transducers, based on a particular and systematic use of the operation of composition in a way that allows for incremental substitutions of concatenated lexical morpheme specifications with their surface realization for non-concatenative processes (the case of Arabic templatic interdigitation and non-templatic circumfixation).

Keywords: Arabic, morphology, non-concatenative, finite state, composition.

1 Introduction

In this paper we propose a new approach to the description of Arabic morphology using 2-tape finite state transducers, based on a particular and systematic use of the operation of composition in a way that allows for incremental substitutions of concatenated lexical morpheme specifications with their surface realization for non-concatenative processes (the case of Arabic templatic interdigitation and non-templatic circumfixation). Then we compare it with what in our opinion represents the state-of-the-art among the 2-tape finite-state implementations, that of Xerox [1], which is mainly based on the operation of intersection. We intentionally limit ourselves to the evaluation of 2-tape strictly finite-state implementations for this paper, leaving out n-tape implementations such as [2] and [3], and those based on extended finite-state automata, such as [4]. In any case we believe that our approach could be trivially adapted to n-tape implementations as well.

In this paper we argue that:

1. the use of composition allows to overcome certain technical problems inherent to the use of intersection;
2. the method of incremental substitutions through compositions allows for an elegant description of all main morphological processes present in natural languages including non-concatenative ones in strict finite-state terms, without the need to resort to extensions of any sort;
3. our approach allows for the most logical encoding of every kind of dependency, including traditional long-distance ones (mutual exclusiveness), circumfixations and idiosyncratic root and pattern combinations;
4. a smart usage of composition such as ours allows for the creation of a same system that can be easily accomodated to fulfil the duties of both a stemmer (or lexicon development tool) and a full-fledged lexical transducer.

Here below is a short review of the Xerox implementation that we hold as our 'gold standard'.

2 Review of the 'Gold Standard'

The Xerox implementation follows from a late eighties commercial project at ALPNET which resulted in an Arabic morphological analyzer based on an enhanced Two-Level approach [5].

$$a{:}b \,. \tag{1}$$

$$a{:}\varepsilon\ a{:}b \cap a{:}\varepsilon\ a{:}b \,. \tag{2}$$

$$a{:}\varepsilon\ a{:}b \,. \tag{3}$$

$$a{:}\varepsilon\ a{:}b \cap a{:}b\ a{:}\varepsilon \,. \tag{4}$$

Two-Level morphology [6] differs from pure finite-state morphology [7] mainly for the way symbol pairs **(1)** are treated: as simple atomic symbols in Two-Level calculus and as relation between languages in the case of pure finite-state calculus. This fundamental distinction determines also difference in behaviour, such as the conception and closure of operations like intersection. Whereas in finite-state calculus intersection of regular language relations is not closed under the set of regular language relations itself, and therefore will not usually be computable, in Two-Level calculus this problem does not occur since intersection is intended not as the intersection of regular language relations but as the intersection of regular languages whose symbols just happen to be pairs. By means of example, **(2)** equals **(3)** in Two-Level calculus but is not computable in finite-state calculus, and **(4)** is still computable in Two-Level calculus but yields the empty set whereas its operands taken alone would encode exactly the same relation inside the framework of finite-state calculus.

The main insight leading the whole Xerox Arabic language analyzer development is that root and pattern (and even vocalism) morphemes could be intersected (more or less problematically) to form a proper Arabic word, an insight which dates back to the Two-Level implementation of [8]. With the licensing of the previous project material from ALPNET to Xerox, consequent moving from Two-Level calculus to finite-state calculus has meant leaving behind any hope of being able to intersect relations, having instead to intersect regular expressions which would be in turn mapped in a totally arbitrary way to any other regular expression by means of a crossproduct operator. There is nothing directly leading from the intersected representation to its 'decomposition' in morphemes.

Our implementation expressly tries to avoid this problem. Note that we do that by using 2 tapes only. We will cover these algorithms more in depth later, but for now let's get back to the kind of problems intersection generates, among which a special place is reserved to the fact that the operator of 'general intersection', meaning the operator having the usual set-theoretical semantics and implemented with the general case algorithms that we would always think of when talking of 'intersection', applied to certain morphemic systems such as that stemming from the linguistic analysis of

[9] (the one returned by the Xerox Arabic Morphological Analyzer demo at http://www.xrce.xerox.com/competencies/content-analysis/arabic/), yields very awkward results.

Here's an example: let | represent the union or disjunction operator, double quotes the escape character and `[C]` the language consisting of the union of every Arabic letter that might constitute a root morpheme, i.e. every consonant (the square brackets are just for grouping purposes).

Using the transliteration system of [10] (of which we present a small fragment in the appendix to this paper), in Xerox Finite State Tool (*xfst*) syntax we would notate that as:

```
define C [' | b | t | v | j | H | x | d | "*" | r | z |
s | "$" | S | D | T | Z | E | g | f | q | k | l | m | n
| h | w | y];
```

According to the analysis outlined in [9] (upon which we agree for purposes of fair evaluation in this review), the Arabic verb اِجْتَمَعَ is composed of a morphemic pattern اِـْتَـَـ (where the *tatweel* symbol stands for any root consonant), a root morpheme ج م ع and a suffix َ .

Now let `&` be the intersection operator, `*` the 0 or more times iteration operator (commonly known as "Kleene star") and `?` a symbol representing any symbol (so-called wildcards).

In this case then, the following two regular expressions are equal:

```
read regex [[A i C o t a C a C & ?* j ?* m ?* E ?*] a];
read regex [A i j o t a m a E a];
```

Unfortunately though, in the case of the verb اِقْتَتَلَ, which is analyzed as composed by the same morphemic pattern as the previous verb plus the root morpheme ق ت ل, an analogous expression would not work. Indeed, the following two regular expressions are equal:

```
read regex [[A i C o t a C a C & ?* q ?* t ?* l ?*] a];
read regex [A i q o t a C a l a | A i q o t a l a C a];
```

This happens because the `[t]` in the root morpheme `[?* q ?* t ?* l ?*]` matches the `[t]` inside the pattern morpheme `[A i C o t a C a C]` leaving either the second or the third `[C]` in there unmatched. An initial workaround to this problem in [11] has been the choice to denote root consonants and template consonants differently, that's to say as in the following root consonant regular expression definition, where curly brackets are normal string symbols used to provide the wanted distinction from template consonants:

```
define C "{" [' | b | t | v | j | H | x | d | "*" | r |
z | s | "$" | S | D | T | Z | E | g | f | q | k | l | m
| n | h | w | y] "}";
```

Beware though, compiling the following consequent new relation results heavier on any machine:

```
read regex [[A i C o t a C a C & ?* "{" q "}" ?* "{" t
"}" ?* "{" l "}" ?*] a];
```

The solution to this commercially critical problem that has been thought of at Xerox is that of a new algorithm, called *merge* [12], whose operator `.m>.` takes as input strings of the kind of `[q t l]` and `{AiCotaCaC}`, as in the following example:

```
undefine C

list C ' b t v j H x d "*" r z s "$" S D T Z E g f q k
l m n h w y;

read regex [q t l] .m>. {AiCotaCaC};
```

In this case, the algorithm's behaviour can be correctly resumed as that of an operation which instantiates the actual value of 'class symbols' `C` in the template network from the value of the symbols in the string described by the filler network, in their correct order; both the template network's language class symbols and the filler network's language symbols must be in the same quantity for the operation to be successful. Note the lack of Kleene stars in the filler network. The same result, with a pure finite-state intersection operation would have been obtained only by compiling the following computationally more expensive expression, where `.o.` is the composition operator, `0` represents an ε-transition and `\"{"` is equivalent to `[? - "{"]` in Xerox syntax:

```
unlist C

define C "{" [' | b | t | v | j | H | x | d | "*" | r |
z | s | "$" | S | D | T | Z | E | g | f | q | k | l | m
| n | h | w | y] "}";

read regex [\"{" | 0:"{" ? 0:"}"]* .o. [[A i C o t a C
a C & ?* "{" q "}" ?* "{" t "}" ?* "{" l "}" ?*] a] .o.
[\"{" | "{":0 ? "}":0]*;
```

To complete the picture concerning the problems generated by the usage of an intersection approach, let's explain the solution adopted at Xerox to resolve the issue we have hinted at some lines ago, that's to say the one related to the impossibility to intersect relations in a finite-state framework. Well, as we have already said, the only possible way to tackle this problem without renouncing to an intersection-based approach is to intersect mere expressions and then turn them into relations by means of a crossproduct operation that maps strings together. [3] though made notice of the fact that in this way the intersection operator becomes a 'destroying' operator, meaning that after one has intersected all the morphemes there is no way to map each one of them to its correct lexical counterpart but instead all one can do is match full superficial (or intermediate, for all that counts) strings like "Aiqotatal" (representing what we may call a ‘stem’, i.e. a pattern still lacking the suffix but correctly fulfilled by the root) to another one (that we can hardly call ‘lexical’ from a linguistic point of view) such as “^[q t l .m>. {AiCotaCaC}^]”.

One might say that this sort of mapping kind of defeats the purpose of building a machine to run the analysis in the first place, if one has to produce the stem analysis by hand. This is not exactly the case though when using the Xerox tools, thank their compile-replace algorithm in [12], which lets the user specify a regular expression whose denoted string gets mapped to a string equal to the regular expression itself. This might sound complicated, but just think of a relation of strings with regular expression syntax, the surface of which will eventually get compiled and represents in fact the string the expression denotes.

Therefore, specifying a relation such as:

```
"^[" "q t l .m>. {AiCotaCaC}" "^]" .x. "^[" "q t l .m>.
{AiCotaCaC}" "^]"
```

(or even shorter, building an expression `"^[" "q t l .m>. {AiCotaCaC}" "^]"` that will be interpreted in proper context as the same identity relation) means in fact to account for the following relation:

```
"^[" "q t l .m>. {AiCotaCaC}" "^]" .x. [A i q o t a t a
l]
```

because one side of the relation gets doubly compiled at will. It becomes a matter of what we may call ‘meta-strings’ at the end (or ‘meta-expressions’, i.e. expressions *about* expressions). Note though that there are some usages of compile-replace different than this that allow for the resolution of problems which would normally exceed finite-state power, such as palindrome extraction, as explicitly stated in [12] itself.

Apart from all these solutions that might not sound very orthodox to formal language theorists and linguists alike, the last big problem with the Xerox implementation is that the lexical analysis in some cases appears to be wrong. We will not try to guess why this might have happened, it could have been the difficulty of encoding proper phonological rules or mere theoretical dissent or anything again, but as a matter of fact every Arabist would tell you that weak verbs of the kind of “qAla” have a lexical origin “qawala” and not “qawula”, even if we all wished it was “qawula” because mapping the superficial string “qulotu” to a lexical counterpart “qawulotu” instead of “qawalotu” would mean easier mapping rules and so on. Unfortunately, that’s not the case [13]. How can we be so sure of this? Well, that’s because of two reasons [14]:

1. verbs with pattern CaCuCa are always intransitive;
2. names deriving from verbs with pattern CaCuCa usually show the pattern CaCiyC (such is the case for instance of “Zariyf” from “Zarufa” and “$ariyf” from “$arufa”) whereas names deriving from verbs like “qAla” show the pattern CACiC (“qa}il”, with *hamza* instead of *waaw* because of other phonological rules)[1].

[1] In Arabic script ظَريـف from ظَرُفَ, شَريـف from شَرُفَ and قـائِل from قـالَ.

Now we show how our implementation avoids many of the aforementioned problems and reaches descriptive elegance without resorting to extensions of any sort.

3 The "Incremental Substitutions" Compositional Approach

Let's have a look at a simplified version of a typical relation encoding in our implementation, using xfst syntax for purposes of consistency throughout our paper.

```
define C [' | b | t | v | j | H | x | d | "*" | r | z |
s | "$" | S | D | T | Z | E | g | f | q | k | l | m | n
| h | w | y];

read regex [[q t l| k t b| T r q] " Form_I_Impf_Act_u"]
.o. [C 0:o C 0:u C " Form_I_Impf_Act_u":0];
```

From an 'analytical' (as opposed to 'generative') point of view we can interpret this last regular relation as a two-phase mapping:

1. in `[C 0:o C 0:u C " Form_I_Impf_Act_u":0]` the vowels in the Verb Form I Imperfect Active pattern ـْـُـ get 'filtered' in the passage from surface to lexical representation, 'erased' and 'substituted' by the agreeing tag which is in fact concatenated to the end of the remaining lexical material made up of those `[C]` roots which were allowed to 'pass through';
2. the resulting lexical string is 'passed' as an argument to a second regular expression `[[q t l | k t b | T r q] " Form_I_Impf_Act_u"]` by means of composition, which will operate on the remaining material if and only if the tags (in this case only 1) concatenated at the end of the regular expression correspond to those generated in or passed through the previous phase of analysis; in this case all it would do on the remaining material would be constraining its quality to that of the actual root morphemes which are allowed to combine with the pattern represented by the concatenated tag.

Notice that with this approach we don't need to previously define the `[C]` language, even if we did it in the previous example. Indeed the following regular expression denotes exactly the same relation as the previous one.

```
read regex [[q t l| k t b| T r q] " Form_I_Impf_Act_u"]
.o. [? 0:o ? 0:u ? " Form_I_Impf_Act_u":0];
```

The employment of ε-transitions is crucial here, because it's the erasure of already analyzed sectors of a string that allows for the smart resolution of the problem of the root and template consonants ambiguity without the need for any additional mechanisms such as those featured in the Xerox implementation, as one can easily ascertain by looking at the following regular expression construct:

```
read regex [[q t l | k t b] " Form_VIII_Impf_Act"] .o.
[? 0:o 0:t 0:a ? 0:i ? " Form_VIII_Impf_Act":0];
```

As stated at the beginning of this paper, we managed to encode a certain kind of dependency (idiosyncratic root and pattern combinations) without the need for unification-based grammars or formalisms and enhancements of any sort. Of course we could not enlist every root and pattern combination in this paper, but by the following expression we show how our implementation organizes more idiosyncratic combinations together in one compact structure:

```
read regex [
[[k t b | q t l] " Form_I_Perf_Act_a"] |
[[D r b | H s b] " Form_I_Perf_Act_i"] |
[["$" r f | H s n] " Form_I_Perf_Act_u"]
] .o. [
[? 0:a ? 0:a ? " Form_I_Perf_Act_a":0] |
[? 0:a ? 0:i ? " Form_I_Perf_Act_i":0] |
[? 0:a ? 0:u ? " Form_I_Perf_Act_u":0]
];
```

Let's now have a look at how circumfixation is handled in our implementation (again, for matters of simplicity and space not all the circumfixes will be enlisted, but of course in the actual implementation items are expanded to fully cover the language):

```
read regex
[[q t l] " Form_I_Impf_Act_u"
[" 1_Pers_Sing_Ind_a" | " 1_Pers_Plur_Ind_a"]] .o.
[? 0:o ? 0:u ? " Form_I_Impf_Act_u":0
[" 1_Pers_Sing_Ind_a" | " 1_Pers_Plur_Ind_a"]] .o.
[0:' 0:a ?* 0:u " 1_Pers_Sing_Ind_a":0 |
0:n 0:a ?* 0:u " 1_Pers_Plur_Ind_a":0];
```

Note that other implementations including the Xerox one in [15] usually deal with certain long-distance dependencies through the use of composition, but in a very different way:

1. all the prefixes, stems and suffixes are concatenated together to form every potential combination (even prohibited ones), and prefixes and suffixes are assigned each a distinctive tag;
2. through the use of composition, patterns featuring mutually exclusive tags are explicitly removed from the network.

Our method, on the other hand, just assigns one tag to each circumfix (for other purposes, moreover) and anyway the correct circumfixation is created in one single process instead of total prefixation plus total suffixation and subsequent pruning.

We're now ready to give an interpretation to our approach from a 'generative' point of view as that of an n-phase mapping:

1. in the first regular expression we enlist in a concatenative way all the morphemes (or rather, their lexical representations) which make up a word, in the order in which we should process their 'merging' with the string we obtain at each phase;

2. in the subsequent regular expressions we process their ‘merging’ with any intermediate string previously obtained, according to the order of the remaining tags at each point, ‘erasing’ one tag at a time after its surface counterpart has been created and merged to the rest.

In this way we were able to give a linear rendering of what globally assumes the entity of a hierarchical representation (cfn. ‘morphosyntax’) or incremental creation of bigger building blocks from already elaborated ones, i.e.:

ق ت ل + ــــْــُـ = قْـتُـل (5)

قْـتُـل + يَـــــُـ = يَـقْـتُـلُ (6)

To conclude this section, let us show how our initial commitment of implementation flexibility is honoured in the following regex:

```
read regex [
[? ? ? " Form_I_Perf_Act_a"] |
[? ? ? " Form_I_Perf_Act_i"] |
[? ? ? " Form_I_Perf_Act_u"]
] .o. [
[? 0:a ? 0:a ? " Form_I_Perf_Act_a":0] |
[? 0:a ? 0:i ? " Form_I_Perf_Act_i":0] |
[? 0:a ? 0:u ? " Form_I_Perf_Act_u":0]
];
```

This is one of the previously shown regexes, only without constraints on the allowed root morphemes for each pattern. By running this kind of machine on an Arabic text input we get an output of all the encountered root bundles classified by the patterns they were found in. This has helped us build the actual lexicon out of different sources.

4 Implementation Evaluation

For purposes of evaluation we have written a script composing more than 4700 root morphemes with the verbal patterns they can actually combine with extracted from several databases, including those in [10]. This grammar compiled in real time on an Intel Pentium M 730 1.60 GHz based Microsoft Windows XP system using the Xerox Finite-State Tool version 2.6.2.

5 Conclusions

In this paper we have explored the potential in a particular encoding of lexical transducers that we have labelled as the “incremental substitutions” compositional model as applied to the Arabic language.

We’ve shown how this new approach solves some technical problems that rise with the intersectional approach used by another 2-tape implementation, the ‘gold standard' one of [1]. We've also given hands-on details on our implementation, exemplifying how most morphological processes and descriptions are actually dealt with by going through some simplified snippets of code.

Moreover, we have designed more than one way our model could be put to practical usage (stemming, field research and lexicon developing, morphological analysis and generation).

Ultimately, we have shown that our model allows for a fair description of Arabic morphology in a strictly finite-state framework without the need to resort to enhancements or extensions of any sort.

References

1. Beesley, K.R.: Finite-State Morphological Analysis and Generation of Arabic at Xerox Research: Status and Plans in 2001. In: Proceedings of the Workshop on Arabic Language Processing: Status and Prospects. 39th Annual Meeting of the Association for Computational Linguistics, pp. 1–8. Association for Computational Linguistics, Morristown, NJ, USA (2001)
2. Kay, M.: Nonconcatenative Finite-State Morphology. In: Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics, pp. 2–10. Association for Computational Linguistics, Morristown, NJ, USA (1987)
3. Kiraz, G.A.: Multitiered Nonlinear Morphology Using Multitape Finite Automata: A Case Study on Syriac and Arabic. Computational Linguistics 26(1), 77–105 (2000)
4. Cohen-Sygal, Y., Wintner, S.: Finite-State Registered Automata for Non-Concatenative Morphology. Computational Linguistics 32(1), 49–82 (2006)
5. Beesley, K.R.: Computer Analysis of Arabic Morphology: A Two-Level Approach with Detours. In: Comrie, B., Eid, M. (eds.) Perspectives on Arabic Linguistics, III: Papers from the Third Annual Symposium on Arabic Linguistics, Benjamins, Amsterdam, pp. 155–172 (1991)
6. Koskenniemi, K.: Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production. Publication 11. University of Helsinki, Department of General Linguistics, Helsinki (1983)
7. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI, Stanford (2003)
8. Kataja, L., Koskenniemi, K.: Finite-State Description of Semitic Morphology: A Case Study of Ancient Akkadian. In: COLING 1988. Proceedings of the 12th Conference on Computational Linguistics, pp. 313–315. Association for Computational Linguistics, Morristown, NJ, USA (1988)
9. Harris, Z.: Linguistic Structure of Hebrew. Journal of the American Oriental Society 62, 143–167 (1941)
10. Buckwalter, T.: Buckwalter Arabic Morphological Analyzer Version 1.0. LDC Catalog Number LDC2002L49. Linguistic Data Consortium (2002)
11. Beesley, K.R.: Arabic Stem Morphotactics via Finite-State Intersection. In: Benmamoun, E. (ed.) Perspectives on Arabic Linguistics, XII: Papers from the Twelfth Annual Symposium on Arabic Linguistics, Benjamins, Amsterdam, pp. 85–100 (1999)
12. Beesley, K.R., Karttunen, L.: Finite-State Non-concatenative Morphotactics. In: Proceedings of the Workshop on Finite-State Phonology. 38th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Morristown, NJ, USA (2000)

13. Wright, W.: A Grammar of the Arabic Language. Munshiram Manoharlal, New Delhi (2004)
14. Bohas, G., Guillaume, J.P.: Etude des Théories des Grammairiens Arabes. Institut Français de Damas, Damas (1984)
15. Beesley, K.R.: Constraining Separated Morphotactics Dependencies in Finite-State Grammars. In: Karttunen, L., Oflazer, K. (eds.) FSMNLP 1998. Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, Bilkent University, Bilkent (1998)

Appendix: Buckwalter Transliteration System

In our code we use the 'Buckwalter transliteration system' presented in [10], of which, for purposes of clarity, we outline here just a small fragment including the letters most used in this paper whose character significantly differs from any of those corresponding to it within the plethora of other systems employed in academic publications.

Table 1. A partial transliteration of Arabic characters using the Buckwalter system

Arabic character	ئ	ا	ح	ش	ض	ط	ظ	ع	ـْ
Buckwalter transliteration	}	A	H	$	D	T	Z	E	o