Infrastructure for high-quality Arabic typesetting

Yannis Haralambous
Département Informatique, ENST Bretagne
CS 83 818, 29 238 BREST Cedex 3
France
yannis dot haralambous (at) enst dash bretagne dot fr

1 Introduction

This paper presents what we consider to be the ideal[1] (or at least a first step towards the ideal) infrastructure for typesetting in the Arabic script. This infrastructure is based on four tools which have partly been presented elsewhere:

1. the concept of *texteme*;
2. *OpenType* (or *AAT*) fonts. In fact in this paper we will talk about "super-OpenType" which consists in using static OpenType substitutions and positionings in dynamic typesetting;
3. Ω_2 *modules*: transformations applied to the horizontal node list before entering the main loop;
4. an extended version of TeX's line-breaking graph for dynamic typesetting.

Textemes have been presented in [12] and [13]. They are atomic units of text extending the notion of character. A texteme is a collection of key-value pairs, some of which are mandatory (but may have an empty value) and others optional or freely definable by the user. Mandatory keys are "character" (a Unicode position), "font" (a font identifier) and "glyph" (a glyph identifier in the given font). The latter two keys are, in some sense, the common part between textemes and TeX's "character nodes" (quoted because in fact a "character node" contains only glyph-related information). Hence, a first application of textemes is to add character data to "character nodes". In fact there are other types of information which we also add to textemes:

- hyphenation: instead of using discretionary nodes, we add hyphenation-related information into textemes;
- color;
- horizontal and vertical offset of glyph (leaving the abstract box "width $\times$ (depth + height)" unchanged);
- metadata;
- etc.

[1] The author has written several papers on the typesetting of Arabic: [6, 7, 8, 9, 10, 15, 16], and has developed Arabic systems using three different methods: a preprocessor (1990), intelligent ligatures (*ArabiTeX*, using TeX--XeT, 1992) and Ω_1 Translation Processes (Ω_1 distribution and *Al-Āmal*, 1994). The system described in this paper will be the fourth (and hopefully the last) Arabic system developed by the author.

We will see how — due to the internal structure of the Arabic writing system — the concept of texteme happens to be particularly useful for text in the Arabic script.

There is no need to present *OpenType* and *AAT* fonts; the reader can consult [20] or the corresponding Web pages at Microsoft and Apple.

Ω_2 *modules* have been presented in [14]. The well-known technique of Ω_1 Translation Processes (ΩTPs) is applied at a later stage of text processing inside Ω_2: just before the *end_graf* procedure, which is called when the complete list of nodes of a paragraph has been stored in memory, before we enter into the line breaking engine which will examine this list of nodes and insert active nodes at potential line breaks.

Ω_2 will output the horizontal list of nodes, in XML. Since we are using textemes, this horizontal list contains both traditional types of nodes as well as texteme nodes. External processes will then transform these XML data and the result is again read by Ω_2 and replaces the original horizontal list.

Besides the nodes of the horizontal list, Ω_2 also includes in the XML data global information such as font name mapping, the current language, etc.

In the next section we will describe the fourth tool, namely the extended TeX graph.

2 Dynamic typesetting and an extended graph

When TeX processes a horizontal list, inserts active nodes for potential breakpoints, calculates badnesses for each arc of the graph and finally finds the shortest path (where we consider badness as a "distance"), glyphs do not change:

- if glyphs are given by "character nodes" then they are static;
- if they are given by discretionary nodes, then we have two possibilities: when there is no line break we have a single node list (possibly containing one or more glyphs) and when there is a break we have two entirely different lists, the "pre-break" and the "post-break". The choice

depends entirely on the fact whether we break or not;

- if they are given by ligature nodes, once again we have two possibilities: when the ligature is not broken then we have a single static glyph. When the ligature is broken we return to "characters" (or at least to something which is a bit closer to the concept of character, even though it is not exactly a character) and apply the main loop again to the two parts (before and after the break), which sometimes results in new ligatures. But once again each node list obtained that way is unique.

In all three cases glyphs either do not change or their change depends only on a line break close to them.

Dynamic typesetting is a method of typesetting where glyphs can change during the process of line breaking, for reasons which may depend on macrotypographic properties such as justification of the line or of the entire paragraph, or more global phenomena like glyphs on subsequent lines touching each other or to avoid rivers, etc.

Dynamic typography was applied by Gutenberg in his Bibles. He was systematically applying ligatures to optimize justification on the line level. It is very useful for writing systems using words but not allowing hyphenation, like Hebrew (where some letters have large versions without semantic overload, these letters have been used mostly at the end of lines, when printers realized that they are facing justification problems) or Arabic.

To perform dynamic typesetting with Ω_2 we are extending the graph of badnesses so that we can have many arcs between two given nodes, each one corresponding to a given width ("width" in the sense of glue, that is a triple — ideal width, maximal stretch, maximal shrink) and to its badness.

Using an extended graph means applying the same principle of optimized typesetting on the paragraph level to paragraphs where glyphs (or glyph groups) have alternative forms (in the case of groups we call them "ligatures"). Performing the calculation of shortest path on such a graph means that the solution will be the best possible paragraph, chosen among all possible combinations of alternate forms of glyphs.

Alas, such a calculation can explode combinatorially. Imagine a paragraph of ten lines, each containing 60 glyphs, that is 600 glyphs in total. Imagine each glyph having two variant forms, that is a total of three choices for each glyph. No ligatures. That would already make $3^{600} \approx 1.87 \cdot 10^{286}$ possible combinations of glyphs, enough for running Ω_2 until the next big bang and beyond, for only ten lines of text ... a perspective which would delight Douglas Adams if he was still with us.

This is why dynamic typesetting requires a *strategy*. Even if choices of glyph variants are mutually independent, one has to define rules to limit the number of glyph combinations.

For example, a realistic strategy could be the following:

1. classify glyphs into N width classes (the higher N the finer the result will be, but the more calculations we will have to do);
2. whenever we have to choose between glyphs in the same class, make a single choice, in a random manner;
3. if we have many choices for which the sum of the widths of classes is relatively constant, choose a single one, randomly.

In other words, we restrict the number of choices to those that give us different widths on the word level. Whenever we have different glyph combinations producing the same global width, we use a random generator to choose a single combination. This strategy is useful when there is no semantic overload. One can imagine refined versions where the combination chosen is not entirely random but is based on more-or-less strict criteria (for example: use ligatures preferably at the end of words, or do not use specific variants in the same word, etc.).

The strategy we have described is based solely on width criteria. But when many different glyph versions are designed the chances that some of them are in conflict (for example, may touch each other) heavily increases. Due again to combinatorial reasons, we can't expect the font designer to anticipate all possible conflicts between glyphs. We need a tool which will test each combination (for example, test whether glyphs are touching) and eventually add an additional penalty to the corresponding arc of the graph.

This tool can work on an interline level so that the calculation of global badness is more complex than just summing up the badnesses of individual arcs. The extended TEX graph used by Ω_2 will use binary arithmetic on flags to add additional badnesses to given path choices.

Up to now, very few fonts exist with extremely many variants (one of which is, for example, *Zapfino* which has ten different 'd' letters) and in the case of calligraphic fonts the document's author (who becomes a "calligrapher") will probably be more interested in (manually) choosing a beautiful combination of glyphs than in having the absolutely best justification by leaving the choice of glyphs to the machine.

The case of Arabic is different: ligatures are much more common (especially in traditional writing styles) and calligraphers have a long tradition of using them in the frame of a justification-oriented strategy (see [3]). This is why the extended graph of TeX will prove especially useful for the Arabic script.

In the following we describe the infrastructure necessary for each step of Arabic text processing.

3 Infrastructure for Arabic text processing

3.1 Preliminaries: Dynamicity of Arabic script

Arabic text justification can be obtained by (in order of priority):

- using blank spaces of variable width (as in other scripts);
- enabling or disabling ligatures;
- choosing between alternative forms of glyphs;
- inserting "keshideh" connections between letters (in systems like [3] and [4] which can produce curvilinear connections; when the "keshideh" is simply a straight line segment, its esthetic value is very doubtful).

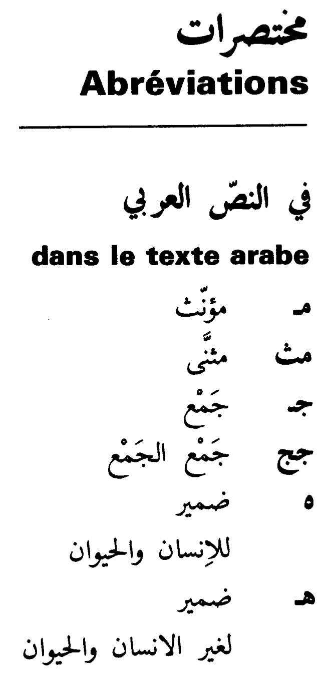
Article [3] mentions some additional justification methods (curvilinear baseline, typesetting the last words as interlinear annotations, etc.), which, in our humble opinion, are less suited for the visual paradigm of printed text and fall into the category of manuscript constructions.

3.2 Preliminaries: Texteme properties

The Arabic script functions in such a way that a lot of information is unwritten and has to be known in advance by the reader:

- Some letters are systematically connected and hence can take up to four different forms according to their immediate context (in Urdu script we have up to nine contextual forms). The form of a glyph can be calculated by contextual analysis, but in some cases this calculation must be overridden: for example, as we see in fig. 1, an initial letter *meem* مـ is used as an abbreviation for *mou'annith* مؤنّث (= female). This contradicts contextual rules: that *meem* normally should be isolated. Unicode provides a solution for this case: to use a ZERO-WIDTH JOINER character just after the *meem*. We consider Unicode's solution to be particularly clumsy: a "character" is (according, once again, to Unicode) the "abstract representation of a smallest component of written language" and this can hardly be said for the ZERO-WIDTH JOINER as used here. Furthermore, when doing copy-and-paste operations one should be sure to copy that invisible character, otherwise the *meem* will change form. Instead, Ω_2 inserts Arabic contextual form inside the texteme, as a texteme property.

  In operating systems there is a dedicated library (Uniscribe for Windows, Pango for Linux, ATSUI for Apple) which performs contextual analysis and then transmits the result to an OpenType font. This is why OpenType provides a property for each contextual form (`init` for initial, `medi` for medial, `fina` for final, `isol` for isolated). In our case, the texteme properties play the rôle of OpenType property activators.
- Short vowels, although not always written, can be very useful for NLP (Natural Language Processing) applications (indexing, automatic translation, summarizing, etc.). People like Ahmed Lakhdar Ghazal in proposing a simplification of the Arabic script ([10], [18]) consider nonetheless that vowels should always be explicitly written in Arabic, to avoid ambiguities.



مختصرات
Abréviations

في النصّ العربي
dans le texte arabe

مـ	مؤنّث
مث	مثنًّى
جـ	جَمْع
جج	جَمْع الجَمْع
ه	ضمير
	للإِنسان والحيوان
هـ	ضمير
	لغير الانسان والحيوان

Figure 1: Abbreviations taken from a French-Arabic dictionary [1]. Notice on line 5 letter *meem* in initial form مـ instead of isolated form م which should normally be used since it is not followed by any other letter. The same happens with letter *heh* on line 11.

Tools like Sakhr's *Diacritizer* [19] or other tools described in [17] can provide the missing vowels. One can imagine an Ω_2 module based on one of these technologies for adding textemes for the missing vowels (all vowels are represented by Unicode characters) with a "hidden" property activated.

Some people may consider full vowelization as archaic. It is true that the average Arabic reader does not need vowels to understand a text, except for rare cases (foreign words, etc.), in which it remains customary to use vowels. By using "visually hidden" textemes we can provide linguistically rich text without changing the visual image of the text and hence people's reading practices.

- Most words of Semitic languages are based on three-letter stems called *roots*. To analyze a word morphologically it is mostly sufficient to find its root and to consider the vowels which are added to the three letters and eventual prefixes and postfixes. In particular, being in possession of these data is the ideal condition for proper indexing of Arabic and the first step for pertinent automatic translation.

  Removing prefixes from Arabic words is necessary even for alphabetical sorting: it would be silly to sort *liradzul* under letter 'l' since it means in fact "for (*li-*) a man (*radzul*)" and the lemma should be *radzul*.

  Once again the solution is provided by textemes: using an NLP tool one can attach properties such as "first/second/third letter of semitic root", "prefix", "postfix", etc., to textemes. The sorter/indexer can then use them to operate properly.
- In some cases we are not sure about some of the information described above: ancient Arabic texts have no dots on letters (so that, for example, letters *beh*, *teh*, *theh*, *noon* and (in initial and medial form) *yeh* are written in exactly the same way), and even fewer short vowels or other diacritics. Reading such a text requires significant human interpretation (as in all ancient languages, but even more because of this particular aspect of Arabic script).

  Let us suppose that a scholar considers that a given letter of his manuscript has a 70% chance of being intended by its author as a *beh*, a 29.999% chance of being a *teh* and a chance in a million to be a *theh*. How can we insert this information into the text itself? Once again, we can use texteme properties. The same method can be applied for missing vowels. And one could develop various visual strategies for representing such textemes (color, hypertext links with pop up windows, etc.).
- Let us leave the semantic area aside and delve again into purely visual issues. Arabic letters can be connected by curvilinear segments called *keshideh*. But the amount of keshideh authorized between two letters is specified by rules (see [5], [9] and [16]). A special texteme property can be used to store the amount of keshideh allowed (for example, a floating number between `0` and `1` or a glue field) after a given texteme.

  Let us note the fact that keshideh has sometimes a semantic overload: it can be used to emphasize or to show metric properties of verses in poetry. In that case the glue field of the keshideh property will have a non-zero ideal value with shrink and stretch values.

  This property is important for post-processing: not only do we need a curvilinear stroke but the glyphs surrounding the keshideh in some cases get modified so that they smoothly fit together.

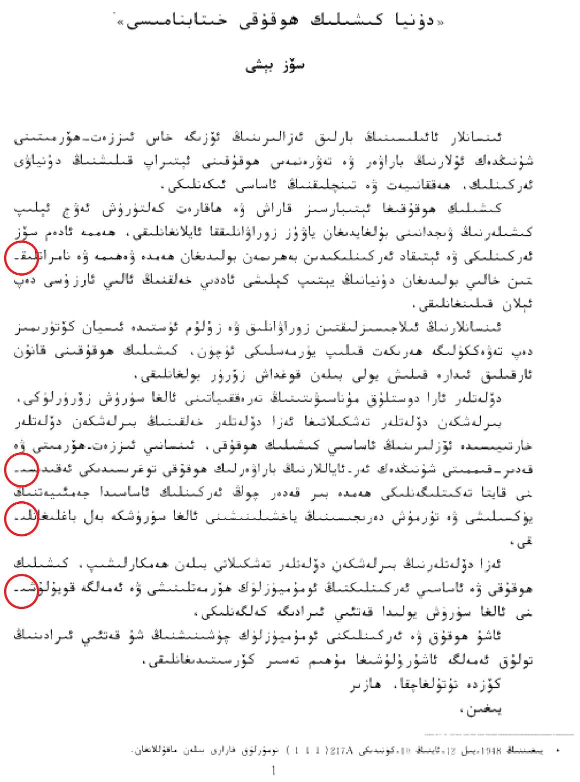
  Last but not least there is another aspect of keshideh: in some cases they may carry short vowels or diacritics. Indeed there is a Unicode character for keshideh: ARABIC TATWEEL. This Unicode character can be used as the base character for short vowels or diacritics (which are all combining characters). In that case we will still use curvilinear keshideh but, again, with a non-zero ideal width so that there is room enough to place the diacritic.

We have presented several cases where injecting extra information into Arabic textemes can prove useful for Ω_2 processing or for other tasks (sorting, indexing, etc.). The other advantage of textemes is that this information will remain in the textual data and will be available to any subsequent operation.

In any case during step 1 of the process we need to calculate Arabic contextual forms. This is described next.

3.3 Step 1: Hyphenation

It has been said over and over again that Arabic is not hyphenated. This is true when we refer to Arabic language, but false when we refer to Arabic script. Indeed, there is one language written in Arabic script, namely Uighur, which uses hyphenation just like any European language. Uighur may use the Arabic script but is not a Semitic language and



«دۇنيا كىشىلىك ھوقۇقى خىتابنامىسى»

سۆز بېشى

ئىنسانلار ئائىلىسىنىڭ بارلىق ئەزالىرىنىڭ ئۆزىگە خاس ئىززەت-ھۆرمىتىنى
شۇنىڭدەك ئۇلارنىڭ باراۋەر ۋە تەۋرەنمەس ھوقۇقىنى ئېتىراپ قىلىشنىڭ دۇنياۋى
ئەركىنلىك، ھەققانىيەت ۋە تىنچلىقنىڭ ئاساسى ئىكەنلىكى.
كىشىلىك ھوقۇقىغا ئېتىبارسىز قاراش ۋە ھاقارەت كەلتۈرۈش ئەۋج ئېلىپ
كىشىلەرنىڭ ۋىجدانىنى بۇلغايدىغان ياۋۇز زوراۋانلىققا ئايلانغانلىقى، ھەممە ئادەم سۆز
ئەركىنلىكى ۋە ئېتىقاد ئەركىنلىكىدىن بەھرىمەن بولىدىغان ھەمدە ۋەھىمە ۋە نامراتلىق-
تىن خالىي بولىدىغان دۇنيانىڭ يېتىپ كېلىشى ئاددىي خەلقنىڭ ئالىي ئارزۇسى دەپ
ئېلان قىلىنغانلىقى.
ئىنسانلارنىڭ ئىلاجسىزلىقتىن زوراۋانلىق ۋە زۇلۇم ئۈستىدە ئىسيان كۆتۈرمىسز
دەپ تەۋەككۈلىگە ھەرىكەت قىلىپ يۈرمەسلىكى ئۈچۈن، كىشىلىك ھوقۇقىنى قانۇن
ئارقىلىق ئىدارە قىلىش يولى بىلەن قوغداش زۆرۈر بولغانلىقى.
دۆلەتلەر ئارا دوستلۇق مۇناسىۋەتنىڭ تەرەققىياتىنى ئالغا سۈرۈش زۆرۈرلۈكى.
بىرلەشكەن دۆلەتلەر تەشكىلاتىغا ئەزا دۆلەتلەر خەلقىنىڭ بىرلەشكەن دۆلەتلەر
خارتىيىسىدە ئۇلارنىڭ ئاساسىي كىشىلىك ھوقۇقى، ئىنساني ئىززەت-ھۆرمىتى ۋە
قەدىر-قىممىتى شۇنىڭدەك ئەر-ئاياللارنىڭ باراۋەرلىك ھوقۇقى توغرىسىدىكى ئەقىدىسى-
نى قايتا تەكىتلىگەنلىكى ھەمدە بىر قەدەر چوڭ ئەركىنلىك ئاساسىدا جەمئىيەتنىڭ
يۈكسىلىشى ۋە تۇرمۇش دەرىجىسىنىڭ ياخشىلىنىشىنى ئالغا سۈرۈشكە بەل باغلىغانلى-
قى،
ئەزا دۆلەتلەرنىڭ بىرلەشكەن دۆلەتلەر تەشكىلاتى بىلەن ھەمكارلىشىپ، كىشىلىك
ھوقۇقى ۋە ئاساسىي ئەركىنلىكنىڭ ئومۇميۈزلۈك ھۆرمەتلىنىشى ۋە ئەمەلگە قوبۇلۇش-
نى ئالغا سۈرۈش يولىدا قەتئىي ئىرادىگە كەلگەنلىكى.
ئاشۇ ھوقۇق ۋە ئەركىنلىكنى ئومۇميۈزلۈك چۈشىنىشنىڭ شۇ قەتئىي ئىرادىنىڭ
تولۇق ئەمەلگە ئاشۇرۇلۇشىغا مۇھىم تەسىر كۆرسىتىدىغانلىقى.
كۆزدە تۇتۇلغاچقا، ھازىر
يىغىن،

* يىغىنىنىڭ 1948-يىل 12-ئاينىڭ 10-كۈنىدىكى 217A (III) نومۇرلۇق قارارى بىلەن ماقۇللانغان.

1

Figure 2: A text in Uighur (Universal Declaration of Human Rights).

hence does not use implicit short vowels: all vowels are explicitly written and one can easily identify syllables and hyphenate words between them.[2]

Uighur is indeed hyphenated but if we add soft hyphen characters we risk obstruction of contextual analysis, which is the next step. It is easier to add potential breakpoints as texteme properties, as is done for other languages.

3.4 Step 2: Contextual analysis

This is the step where the forms of Arabic letters are calculated based on the context. In the (rather rare) case where the user may want to manually specify a contextual form he/she just needs to set the corresponding texteme feature and to lock it, so that the contextual analysis module will not be able to change it.

Contextual analysis should be done on the module level rather than on the ΩTP level because ΩTPs transform only the contents of a buffer, so that any command inserted into an Arabic word will end the buffer and will obstruct contextual analysis. For example, if we want to colorize a part of an Arabic word we would normally use a `\textcolor` command, but this would enter in conflict with contextual analysis and the colorized letter would be typeset always in isolated form.

Performing contextual analysis on the level of nodes of the horizontal list allows us to obtain a much more reliable result.

3.5 Step 2′: Hamza rules

One of the major contributions of ArabTEX (by Klaus Lagally) to Arabic text processing was the fact that it considers Arabic script from a grammatical point of view, while Unicode is bound, by its tenth general principle, to follow legacy encodings like ASMO, which in turn have been based on the character set of Arabic typewriters.

There is one case where the difference between ArabTEX and Unicode is flagrant: the use of *hamza*. This letter represents the glottal stop and can be represented in four possible ways: as an isolated letter (ARABIC LETTER HAMZA), or carried by *alef* (ARABIC LETTER ALEF WITH HAMZA), by *yeh* (ARABIC LETTER YEH WITH HAMZA), or by *waw* (ARABIC LETTER WAW WITH HAMZA). Four Unicode characters for what is in fact a single letter.

The rules of visual representation of *hamza* are quite strict:

1. at word begin: *hamza* is carried by *alef*;
2. inside a word:
   (a) if the *hamza* is preceded or followed by a vowel /i/ (short or long), it is carried by *yeh*,
   (b) otherwise, if it is preceded or followed by a vowel /u/ (short or long), it is carried by *waw*,
   (c) if it is preceded by *yeh* and followed by short vowel /a/, it is carried by *yeh* (with some isolated exceptions where it is carried by *alef*),
   (d) if rules (a), (b), (c) do not apply and it is preceded or followed by a short vowel /a/ it is carried by *alef*,
   (e) if rules (a)–(d) do not apply and it is preceded by a long vowel /a/ it is written without carrier;
3. at word end: if it is preceded by a short vowel /i/, /a/ or /u/ then it is carried by the corresponding long vowel; if it is preceded by a long vowel or a consonant, then it is written without carrier.

[2] It seems that one of the reasons why hyphenation is not desirable for Ivrit (although used by Israeli newspapers) is the fact that the absence of short vowels may lead to bad hyphenations, a problem which could be solved by vowelization.

ArabTEX indeed takes an abstract representation of *hamza* as input and calculates the visual representation according to the rules (and exceptions) above. A "hamza module" for Ω_2 could serve either to facilitate input of Arabic text (but in that case one should develop the corresponding GUI) or as a "spelling checker" for the specific grammatical issue.

3.6 Step 3: Bidi algorithm

The bidi algorithm is part of the Unicode specification. It deals with the visual representation of mixed RL (right-to-left) and LR (left-to-right) text. For example, if we consider capitals to be Arabic, when typesetting the sentence "My friend said شكراً!" must the exclamation mark be placed to the right or to the left of شكراً? In other words: is this exclamation mark part of the Arabic sentence "شكراً" or part of the English sentence "My friend said [...]!"? In the first case ("Arabic exclamation mark") the exclamation mark is placed "after" شكراً, and "after" in Arabic means "to the left of it". In the other case it is placed to the right of شكراً.

The problem here is that the exclamation mark has no explicit directionality: it may be equally well considered as being RL or LR. The bidi algorithm gives a canonical default solution to this problem and more generally to the way of rendering a paragraph containing LR, RL as well as neutral (with respect to directionality) characters.

One may ask: "why does Unicode care about rendering issues?" The reason is that one does not always want the canonical solution. To change the order in which blocks of the paragraph are displayed, one can use special Unicode characters RLE, LRE, RLO, LRO, PDF, RLM, LRM. This may sound complicated but in the everyday life of an Arabic language keyboard user, whenever a paragraph does not look like he/she expected it, he/she only needs to insert a character or two among these to obtain the correct rendering. The bidi algorithm is applied on-the-fly by WYSIWYG systems.

What happens when such a text is processed by Ω_2? The latter just needs to perform the same calculations as to obtain the same results as the WYSIWYG system. To do this, one needs to consider the paragraph as a whole. And this is only possible on the level of the horizontal list. This is why a separate bidi module must be applied, and this is step 3 of the process.

3.7 Step 4: OpenType and super-OpenType features

In an Arabic rendering process OpenType tables can fulfill five functions:

1. Supply the glyph corresponding to the pair (character, contextual form), the form being provided as an OpenType feature [GSUB table, lookup of type 1 "single substitution"];
2. Supply grammatical (*lam-alif*) and esthetic ligatures [GSUB table, lookup of type 4 "ligature"];
3. Supply alternative forms for glyphs [GSUB table, lookup of type 3 "variant selection"];
4. Kerning between single or ligatured glyphs [GPOS table, lookup of type 2 "positioning of a pair of glyphs"];
5. Place short vowels and other diacritics on isolated glyphs or on ligature components [GPOS table, lookups of types 4 "diacritical marks" and 5 "diacritical marks on ligatures"].

One could imagine some of these features being contextual, with or without backtrack and lookahead. One could also imagine an Arabic OpenType font using the lookup of type 3 "cursive attachment" of the GPOS table.

Clearly these lookups handle most of the complexity of Arabic script. In Ω_2, the GSUB and GPOS tables of the font are read by corresponding modules, which will transform the horizontal list of textemes accordingly.

More precisely, on a first parse of the font we store the glyphs which start a context or match a lookup, as well as the maximum length of context (with backtrack and lookahead) for each glyph. Then, when going through the horizontal list of textemes, for each glyph of a texteme we test whether it is part of a context and then check the following glyphs up to the maximum length for that glyph.

According to OpenType rules we must stop at the first lookup which matches the longest string.

What we call "super-OpenType" is the fact that we take not only the longest string but also all substrings starting by the same glyph, and also that we process all possible lookup matches and not only the first in the list. By doing this we store all possible OpenType transformations so that they produce distinct arcs in the TEX paragraph builder graph. This behaviour is only valid when we are doing dynamic typesetting.

For example, one can imagine four glyphs $g_1g_2g_3g_4$ forming a ligature (like the Arabic word "muhammad"). In a standard OpenType process, the application would find this ligature and stop. In super-OpenType we also store all possible "subligatures" in our texteme: $g_1g_2g_3$, g_1g_2, g_3g_4, etc. as well as single unligatured glyphs g_1, g_2, g_3, g_4. Every choice of ligatures and/or single glyphs results in a different badness calculation for the given line, and hence is a different arc of our (extended) graph.

The reader may wonder how we deal with hyphenation (although there is only one Arabic script language which is hyphenated: Uighur). As hyphenation is performed before OpenType transformations, we already have "alternative horizontal lists" (also called "bifurcations"): with and without line break. We can consider this a split of the horizontal list into two parts: one goes unaltered, and the other contains the hyphen and a special texteme property representing a line break. The OpenType modules go through both parts and apply the necessary transformations, eventually involving the glyph of the hyphen and special end-of-line or begin-of-line features ([11]).

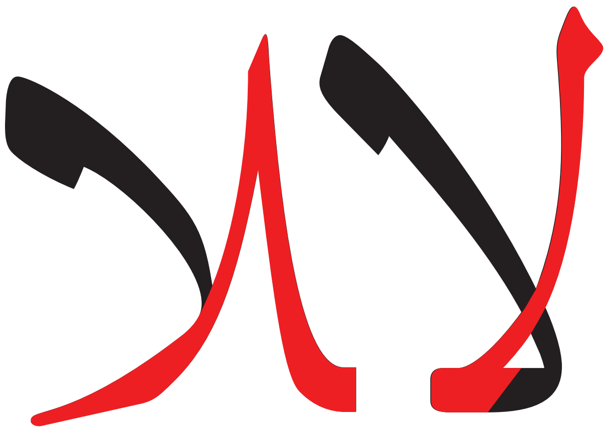


Figure 3: The *lam-alif* ligature as two glyphs.

Let us note *en passant* an important problem of Arabic fonts: ligatures *lam-alif* are always drawn as a single glyph. This is justified by the fact that the two letters are always connected, but can be quite problematic when we want to color one of the two letters. We suggest using the following approach: instead of implementing *lam-alif* as a single glyph obtained by a type 4 lookup, divide the ligature glyph in two parts and implement them as variants of the corresponding glyphs, obtained by type 1 lookups ("single substitutions"). In that way they can be colorized separately, but otherwise the visual result is the same.

This problem occurs only for the *lam-alif* ligature since in all other cases we deal with esthetic (and hence non-mandatory) ligatures and one can simply break the ligature into single and separately colorizable glyphs.

3.8 Step 5: Fine-tuning

An OpenType font designer is, after all, just a human being, and cannot possibly anticipate *all* possible combinations of glyphs, ligatures, short vowels, diacritics. To face this problem, and knowing that automatic positioning of Arabic short vowels and diacritics has been studied extensively [2], one can adopt at least two approaches:

1. Develop a system which will either refine the font tables whenever a conflict appears in a word (for example two letters touching each other, or a diacritic touching a letter, or a diacritic set in such a way that it is not clear to which letter it belongs) or simply break the corresponding ligature and return to a non-ligatured state (where no conflicts occur);
2. Develop a system checking the default (OpenType) positioning and correcting it accordingly (by slightly moving some of the visual components). This solution is especially interesting when one does not have the rights to modify the font.

This step deals with the latter solution: correcting *a posteriori* the positioning of Arabic letters, short vowels and diacritics, obtained by OpenType transformations. This involves heavy, but well understood, calculations based on glyph outlines: obtain the glyph outlines, place them at given horizontal and vertical offsets, and check whether they touch or even come closer than a given ε to each other (with a lot of special cases depending on optical effects).

Another solution would be to take pixel images of the various glyphs *emboldened* (so that we also catch glyphs getting close without touching), assemble them using the corresponding horizontal and vertical offsets, and find pixels belonging to more than one glyph.

As always in such cases, a tool as the one described would need serious optimizations to run without slowing down the whole typesetting process. One could imagine a cache mechanism for storing words that present such problems so that the system does not need to do redundant calculations.

There are several ways to proceed for solving this problem. What we wish to point out is the fact that in an infrastructure as the one described here, it is possible to take control of glyphs *after* they have been transformed by OpenType rules.[3]

3.9 Step 6: Extended graph and strategies

We arrive now at the paragraph builder. After having gone through the five steps described above,

[3] At this level we recommend operating only on the glyph level. Indeed it would be unwise to change character values of textemes, but, at the very end, this is up to the user to decide: he/she has full control over the textemes before they are transmitted to the paragraph builder.

we have a horizontal list with, occasionally, variant glyphs, some of which are logically connected (for example, when we have a line break). This means that we have several ways of wandering across the horizontal list, according to our choices of variant glyphs and/or line breaks. As in standard TEX we start defining active nodes and calculate the badness of (potential) lines brought to the same width. But in the extended graph we have many graph arcs sharing the same departure and arrival nodes: as many as there are combinations of variant glyphs and activated/de-activated ligatures.

The extended graph is larger than the ordinary TEX graph, but the process of calculating the shortest path is the same. Nevertheless we may have to include logical expressions based on flags corresponding to arcs so that a given path through the graph may accumulate an extra penalty because of interline phenomena (lines touching each other, rivers, etc.).

As already mentioned in the introduction, when a font provides a large number of glyph variants, the badness calculation will suffer from combinatorial explosion. To avoid this we need to implement a *strategy* before entering into the extended graph. The goal of the strategy is to significantly decrease the number of arcs between two given nodes, and nevertheless obtain the smallest possible badness.

This is possible since the badness of a line depends only on widths of glyphs and not on the glyphs themselves. Which means that if we consider classes of glyphs of the same width we obtain the same result and hence need to perform our calculations only on the class level. Once the optimal classes have been found, one can simply choose glyphs randomly in the same class.

But even when using classes of glyphs one can obtain the same badness by ordering them differently. For example, a word can be typeset by using a "wide" class followed by a "narrow" one, or the other way around: the global width will be the same, and so will be the badness. The next step of the strategy would be to choose patterns of classes or at least eliminate arcs based on the same glyph classes but in different orders.

Let us not forget that besides the combinatorics of "rigid" widths provided by variant glyphs, we also have glue, obtained by blank spaces as well as by keshideh. The precision of classes of glyph widths must be set in inverse relation to the amount of glue we can use. The more keshideh and interword glue we will use the less precision we need since differences in width between the glyph class and the glyph actually used will be absorbed by glue.

3.10 Step 7: Post-processing

Multiple master tables have been defined for OpenType and then dismissed, as the Multiple Master format is now officially obsolete. AAT variation tables have been used only in a single experimental font (*Skia*). Graphite does provide continuous variation of glyphs. In one word: there is nowadays no glyph variation in font formats.

How do we deal then with keshideh and other glyph variations which have to be continuous?

A possible solution would be to use a post-processor: a *dvips* module which will convert glue (flagged as "keshideh glue") into beautiful curvilinear strokes and replace the glyphs surrounding the keshideh by appropriate variants.

One could imagine a continuous stroke generator for generating keshideh but also a large set of predesigned keshideh and surrounding glyphs. The latter solution has the advantage of using hints, that can be useful at low resolutions, and should accelerate processing.

3.11 Final result

At the end we obtain textemes containing all the information accumulated through these seven steps, as well as the initial information: Unicode characters, contextual forms, semitic roots, full vowelization, etc.

References

[1] *Mounged de poche.* Dar el-Machreq, 1991.

[2] Gábor Bella. An automatic mark positioning system for Arabic and Hebrew scripts. Master's thesis, ENST Bretagne, October 2003.

[3] Mohamed Jamal Eddine Benatia, Mohamed Elyaakoubi, and Azzeddine Lazrek. Arabic text justification. *TUGboat*, 27(2):137–146, 2006.

[4] Daniel M. Berry. Stretching letter and slanted-baseline formatting for Arabic, Hebrew and Persian with ditroff/ffortid and dynamic PostScript fonts. *Software Practice and Experience*, 29(15):1417–1457, 1999.

[5] Carl Faulmann. *Das Buch der Schrift enthaltend die Schriftzeichen und Alphabete aller Zeiten und aller Völker des Erdkreises.* Druck und Verlag der kaiserlich-königlichen Hof- und Staatsdruckerei, Wien, 1880.

[6] Yannis Haralambous. Arabic, Persian and Ottoman TEX for Mac and PC. *TUGboat*, 11(4):520–524, 1990.

[7] Yannis Haralambous. Towards the revival of traditional Arabic typography. In *Proceedings of the 7th European TEX Conference, Prague*, pages 293–305, 1992.

[8] Yannis Haralambous. Typesetting the Holy Qur'ān with TEX. In *Proceedings of the 3rd International Conference and Exhibition on Multilingual Computing, Durham 1992*, 1992.

[9] Yannis Haralambous. The traditional Arabic typecase extended to the Unicode set of glyphs. *Electronic Publishing—Origination, Dissemination, and Design*, 8(2/3):111–123, 1995.

[10] Yannis Haralambous. Simplification of the Arabic script: Two different approaches and their implementations. In *Electronic Publishing, Artistic Imaging and Digital Typography*, volume 1375 of *Springer Lecture Notes in Computer Science*, pages 138–156. Springer, 1998.

[11] Yannis Haralambous. New hyphenation techniques in Ω_2. *TUGboat*, 27(1):98–103, 2006.

[12] Yannis Haralambous and Gábor Bella. Injecting information into atomic units of text. In *Proceedings of the ACM Symposium on Document Engineering, Bristol*, 2005.

[13] Yannis Haralambous and Gábor Bella. Omega becomes a sign processor. *EuroTEX 2005*, pages 99–110, 2005.

[14] Yannis Haralambous and Gábor Bella. Open-belly surgery in Ω_2. *TUGboat*, 27(1):91–97, 2006.

[15] Yannis Haralambous and John Plaice. First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer. *TUGboat*, 15(3):344–352, 1994.

[16] Yannis Haralambous and John Plaice. Multilingual typesetting with Ω, a case study: Arabic. In *Proceedings of the International Symposium on Multilingual Information Processsing, Tsukuba 1997*, pages 137–154. ETL Japan, 1997.

[17] Ruhi Sarikaya Imed Zitouni, Jerey S. Sorensen. Maximum entropy based restoration of Arabic diacritics. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the ACL, Sydney*, pages 577–584, 2006.

[18] Ahmed Lakhdar-Ghazal. *Pour apprendre et maîtriser la langue arabe*. Éditions La Porte, 1991.

[19] Sakhr. Diacritizer. `http://www.sakhr.com/Sakhr_e/Technology/Diacritization.htm`.

[20] Yannis Haralambous (translated into English by Scott Horne). *Fonts & Encodings*. O'Reilly & Associates, 2007.