

Implementation of Reverse Chaining Mechanism in Pango for Rendering Nastaliq Script

Abstract

This paper implements the OpenType's Reverse chaining table in Pango to enable complex Arabic scripts such as Nastaleeq to be rendered correctly in Linux. The OpenType formalism provides a number of glyph substitution tables to handle all forms of glyph switching, including the Type 8 reverse chaining table. This table is useful for modeling Arabic scripts. Although these scripts are written from right-to-left, but when it comes to context, each letter is dependent on the letter to its left (i.e. on the following letter) – resulting in context dependency drift from left-to-right. This behavior can best be modeled by the reverse chaining table. This table was implemented in Pango and has been incorporated into freeware online Pango libraries. Implementation of this table has enabled the modeling of Urdu scripts according to its natural way of writing.

Introduction

Urdu is spoken by more than 60 million speakers in over 20 countries [1]. Urdu is derived from Arabic script. Arabic has many writing styles including Naskh, Sulus, Riqah and Deevani. Urdu however is written in Nastaliq script which is a mixture of Naskh and an old obsolete Taleeq styles. This is far more complex than the others. This cursive font is highly context sensitive. Shape of a letter depends on multiple neighboring characters. In fact, in almost all cases, the shape of the letter depends on the shape of the letter following it. As a result, a letters can have up to 28 shapes each, at medial and initial position [5]. Another interesting property of Nastaliq is the alternating Thick-Thin-Thick joins. This is discussed next in detail.

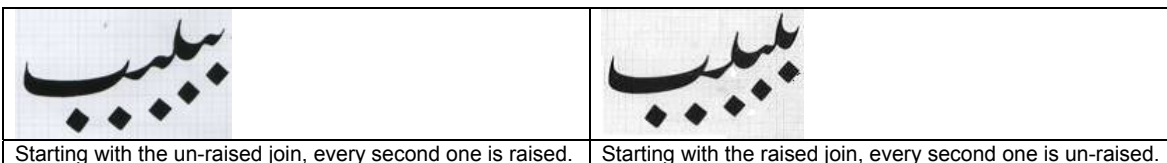
The Alternating Thick-Thin-Thick Joins

It is possible that many similar shapes come together in a ligature, when a same letter is written repetitively. For some letters like *bay*, the similar shape renders it difficult to make out what is written i.e. the perception of the joins becomes difficult. That is why whenever similar cusp-like joins come together, as in *bay*, the monotony of their shape is intentionally broken by differing the shape of alternate joins. When two or more Bay's (or Bay-like joins) come together, every alternate join is raised to make it different from the surrounding ones. This helps in the perception i.e. reading of the Nastaliq text. The principle is illustrated below with some examples.



The raised join is shown in red circle.

As already mentioned above, the raised joins or raised cusps are used to break the monotony of the similar joins when several cusps come together. The frequency of usage of the raised and un-raised joins is almost equal. Hence, a ligature can in principle be formed in either of the two ways: raised joins coming in between un-raised ones and un-raised joins coming in between raised ones. However, in practice, the first one is preferred. The notion behind this is to minimize the number of raised cusp as much as possible.



This characteristic of Nastaliq can be correctly modeled if the text is processed in reverse order instead of the usual first-to-last glyph order, and this is only possible through the reverse chaining table and mechanism. Reverse chaining table and mechanism processes the string

backwards. The advantage of this is that whenever a letter is being processed, the glyph after it has already been processed, and provides the correct context for the current glyph. Since in almost all cases, the shape of the letter depends on the letter following it.

OpenType

OpenType is a font format developed and promoted jointly by Microsoft and Adobe. It extends but not modify, the basic directory-table structure of TrueType. This makes the smart font, a more natural solution for complex text processing. It is a set of tables, which are added to a TrueType font to allow for glyph substitution, glyph positioning, multiple baselines, and justification. The table of interest in this paper i.e. the glyph substitution table or GSUB, provides character to glyph and glyph(s) to glyph(s) mapping options. This is all possible because OpenType provides linguistic information for proper glyph substitution and accurate typographic composition. The GSUB data is organized by script, language system, and typographic feature [3].

Script > Language System > Feature > Lookup

The lookup contains the actual rules. There are two type of lookups. One for substitution and other for positioning. The substitution lookups contains rules from the input character or glyph(s) to output glyph(s), and the context in which the rule should apply. An example of a lookup is given below:

Shape 1 → Shape A
Shape X → Shape New
Context Before: B, Context After: LL

That is shape 1 and X become A and New respectively after B and before LL. If there is another shape that needs to be substituted in the exact same environment, its rule should be written in this lookup instead of another lookup. It can also be said that a lookup actually represents an environment or a context, and different lookups represent different contexts in which glyph substitution can occur.

Pango

Pango is an open-source framework for the layout and rendering of internationalized text in Linux. Pango uses Unicode for all of its encoding, and will eventually support output in all the world's major languages. Pango was designed to be a text layout engine. It encapsulates all the necessary knowledge about various languages and scripts. It handles almost every writing system in the world, and can work on top of multiple display systems – including traditional X fonts, or client-side OpenType fonts[4]. Pango is freeware and available from www.pango.org.

Languages such as Arabic and Hebrew are written from right-to-left, instead of from left-to-right, so the rendering process needs to be able to deal with that ordering. Urdu/Arabic also introduces some other complications. This is the context sensitivity of these languages. In these languages the shape of each character is different depending on whether it occurs in isolation or at the beginning, middle or the end of a word. So there should be some mechanism that could determine the right glyph of the character depending on its position or context within the word.

The Pango library was designed to handle these complexities. It serves as module that holds all the knowledge about various languages and scripts, and utilizes this information to properly layout each language. Thus providing a higher level of abstraction, where the application (e.g. Gedit) simply presents this library with a chunk of text and all the low level details like laying out the text, applying script specific operations on the text, choosing glyphs and rendering it etc. is done by the library.

Text Processing Algorithm

The lookup processing algorithm can best be described in the following word:

“During text processing, a client applies a lookup to each glyph in the string before moving to the next lookup. A lookup is finished for a glyph after the client locates the target glyph or glyph context and performs a substitution, if specified.” (OTF[3])

That is the first lookup is applied on the first glyph in the string. Then on the second glyph and so on till the last one. Then the second lookup is tried in the same way. Then the third lookup and so on. The glyph are processed in the input or logical order. So, for right-to-left scripts, the first glyph in the glyph string is the rightmost one while for left-to-right scripts it is leftmost.

Modeling Using OpenType

To show the working of the text/lookup processing algorithm given above, and how its problematic for Nastaliq, a sequence of *bays* have been modeled in OpenType:



This has six different shapes. Starting from right to left, these can be named as bayInItThick, bayMediThin, bayMediThick, bayMediTiny, bayMediBfBayFinal and FinalBay. Let's try to define rules and lookups for this word and then apply them on the algorithm. Assume that all medial and initial bays already have an assigned shape bayMedi1 and bayInIt1 respectively. The initial configuration is given below:

FinalBay	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayInIt1

The lookups have been arranged in such a way that it 'forces' the text to be processed backwards. For example, if the lookup which has the final *bay* as the context, is placed first, then the second last letter will be processed before all previous letters.

Lookup 1

The first lookup will transform the bay immediately before final bay.

bayMedi1 → bayMediBfBayFinal : whenever it is followed by bayFinal.

As a result we get:

FinalBay	bayMediBfBayFinal	bayMedi1	bayMedi1	bayMedi1	bayInIt1

Lookup 2

The second lookup will transform the third last bay to its correct form.

bayMedi1 → bayMediTiny : whenever it is followed by bayMediBfBayFinal.

The string now looks like:

FinalBay	bayMediBfBayFinal	bayMediTiny	bayMedi1	bayMedi1	bayInIt1

Lookup 3

The third lookup will transform the fourth last bay to its correct form.

bayMedi1 → bayMediThick : whenever it is followed by bayMediTiny

The string now looks like:

FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMedi1	bayInIt1

Lookup 4







The fourth lookup will transform the second bay to its correct form.

bayMedi1 → bayMediThin

Similarly for initial bay we can have

bayInit1 → bayInitThin : whenever it is followed by bayMediThick

Since both bayInit1 and BayMedi1 have the same context, they can be placed in the same lookup. The string now looks like:

					
FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayInit1


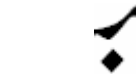
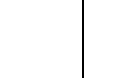
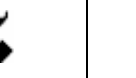
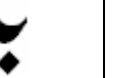

Lookup 5

And finally for the initial bay we have

bayInit1 → bayInitThick

Similarly for medial bay we can have

bayMedi1 → bayMediThick : whenever it is followed by bayMediThin

					
FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayInitThick

Once the lookups are defined, they are processed on the input string that contains 6 bays after the initial, medial and final rule have been applied.

Direction of Processing	←					
Input string 'in'	in[5]	in[4]	in[3]	in[2]	in[1]	in[0]
Pre-calt	FinalBay	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayInit1
After lookup 1	FinalBay	bayMediBfBayFinal	bayMedi1	bayMedi1	bayMedi1	bayInit1
After lookup 2	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMedi1	bayMedi1	bayInit1
After lookup 3	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMedi1	bayInit1
After lookup 4	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayInit1
After lookup 5	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayInitThick

As a result we get the desired output. Now consider the string of length 7, that contains all bays.

Direction of Processing	←						
Input string 'in'	in[6]	in[5]	in[4]	in[3]	in[2]	in[1]	in[0]
Pre-calt	FinalBay	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayInit1
After lookup 1	FinalBay	bayMediBfBayFinal	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayInit1
After lookup 2	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMedi1	bayMedi1	bayMedi1	bayInit1
After lookup 3	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMedi1	bayMedi1	bayInit1
At this stage baymedi1 at second index (in [1]) should also get transformed to bayInitThick, but this did not happen because the context after, as given in lookup 3, for it is bayMediTiny. In this case however it is followed by bayMedi1 so the rule does not apply. The lookup 3 there fore requires modification. Continuing with the remaining lookups we get:							
After lookup 4	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayMedi1	bayInit1
After lookup 5	FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayMediThick	bayInit1

Hence we get an erroneous output.

Modification of Lookup 3

Since the processing of lookups goes from start to end, the glyph at index 1 is processed before glyph at index 3. Accordingly the lookup 3 after modifications become

bayMedi1 → bayMediThick : whenever it is followed by bayMediTiny or
 whenever it is followed by baymedi1 baymedi1 bayMediTiny

This solution leads to another problem. For the word of length 9, the lookup has to be again modified to:

bayMedi1 → bayMediThick : whenever it is followed by bayMediTiny or
 whenever it is followed by baymedi1 baymedi1 bayMediTiny or
 whenever it is followed by baymedi1 baymedi1baymedi1 baymedi1 bayMediTiny

In conclusion, the longer the word, the bigger the context. Clearly such problem could not be addressed with the current contextual substitution support. The implementation of the reverse chain contextual substitution was essential.

Reverse Chain Contextual Substitution Table

Implementation of the Reverse Chain Contextual Single Substitution will enable the modeling of Urdu scripts according to its natural way of writing. The major difference between this and the lookup algorithm defined earlier, is that processing of input glyph sequence goes from end to start. Before moving on to the design and implementation of this table and it working, let us re-look the problem of previous section and how introducing a reverse chain lookup solves the problem. Assuming the existence of all the lookup tables listed in chapter, a small modification to lookup 3 solves the problem indefinitely.

Lookup 3 re-visited

1. Change type of lookup to reverse chain table, which means that this table will be applied on the string from end to start. And, replace all rules with this one:
2. bayMedi1 → bayMediThick
 whenever it is followed by bayMediTiny or
 whenever it is followed by baymedi1 baymediThick

Implementation

The reverse chain lookup type 8 can be incorporated into the current glyph substitution support in three steps. The first step is to include a reverse chain contextual substitution table in the GSUB table which in turn requires a structure for holding the reverse chain substitution lookup tables. The second step is to load these tables and the third step is to apply these lookups on the input glyph string.

Adding Reverse chain substitution table to GSUB table

The reverse chain contextual substitution table as specified by OT is given in table 1 below

Type	Name	Description
uint16	SubstFormat	Format identifier-format = 1
Offset	Coverage	Offset to Coverage table - from beginning of Substitution table
uint16	BacktrackGlyphCount	Number of glyphs in the backtracking sequence
Offset	Coverage[BacktrackGlyphCount]	Array of offsets to coverage tables in backtracking sequence, in glyph sequence order
uint16	LookaheadGlyphCount	Number of glyphs in lookahead sequence
Offset	Coverage[LookaheadGlyphCount]	Array of offsets to coverage tables in lookahead sequence, in glyph sequence order
uint16	GlyphCount	Number of GlyphIDs in the Substitute array
GlyphID	Substitute[GlyphCount]	Array of substitute GlyphIDs-ordered by Coverage Index

Table 1 ReverseChainSingleSubstFormat1 subtable: Coverage-based Reverse Chaining Contextual Single Glyph substitution .

The corresponding structure for the reverse chaining table is given below. FT_UShort is redefinition of unsigned integer (2 bytes). TTO_Coverage is also a data structure for holding the list of glyphs within the lookup. Then there is also a coverage table arrays for lookahead and backtrack sequences.

```
Struct ReverseChainContextSubstFormat1
{
    FT_UShort          SubstFormat,
    TTO_Coverage       Coverage,
    FT_Short           BacktrackGlyphCount,
    TTO_Coverage*      BacktrackCoverage,
    FT_Short           LookaheadGlyphCount,
    TTO_Coverage*      LookaheadCoverage,
    FT_Short           GlyphCount,
    FT_Short*          Substitute,
}

```

The code for loading this data structure is omitted.

Applying Reverse chain substitution table on input string

Once again, the algorithm for processing the lookups

*Starting from lookup LK = 1 to lookupcount
Starting from element IND = 1 to length(input string)
Apply lookup LK on INDth index of input string*

For reverse chaining table, the processing of input glyph string goes from end to start. One solution is to have separate processing mechanism for this table. But this algorithm suffices. The algorithm takes each lookup one at a time and applies it from start to end, but once into the routine that actually applies or processes the reverse chain lookup, the index value is recalculated so that is the exact reflection of the original index value. This was easily calculated using:

$$\text{new_IND} = \text{length}(\text{input string}) - (\text{IND} + 1);$$

Hence the reverse chain lookup is applied on input string from end to start although the algorithm operates from start to end. The relevant part of the routine that implements the manner in which lookups are to be applied is given next.

Routine 4:

```
static FT_Error Do_String_Lookup( TTO_GSUBHeader* gsub,
                                FT_UShort lookup_index,
                                TTO_GSUB_String* in, TTO_GSUB_String* out)

```

```

...
if(gsub->LookupList.Lookup[lookup_index].LookupType ==
                                GSUB_LOOKUP_REVERSE_CHAIN)
{
    new_in_pos = in->length - (in->pos + 1); //INVERTING INDEX
    if ( ADD_String_ReverseOrder( in, 1, out, 1,&s_in[new_in_pos],
                                0xFFFF, 0xFFFF) )
        return error;
}
else
    if ( ADD_String( in, 1, out, 1, &s_in[in->pos], 0xFFFF, 0xFFFF ) )
        return error;
}
...

```

} Note 1

Note 1: If the table is reverse chain, then the indices are reversed.

The routine that implements the manner in which lookups are to be applied is given next. Before moving onto the code, a brief description of the working of the reverse chain lookup on one particular glyph of input string is necessary. The OTF specification: *“When a text-processing client locates a context in a string of text, it finds the lookup data for a targeted position and makes a substitution by applying the lookup data at that location.”*

A generic algorithm following these terms is given below.

1. Match the before context of the lookup with the glyph sequence immediately preceding the current glyph. If match is not made then no further processing is required.
2. Match the current glyph sequence with the glyph sequences in the lookup.
3. Match the after context of the lookup with the glyph sequence immediately following the current glyph. If match is not made then no further processing is required.
4. Apply the substitution rule on glyph and copy the resulting glyph on the output string.

Based on this algorithm the corresponding code is given next.

Routine 5:

```
static FT_Error  Lookup_ReverseChainContextSubst1(TTO_GSUBHeader* gsub,
          TTO_ReverseChainContextSubstFormat1* rccsf1,
          TTO_GSUB_String* in, TTO_GSUB_String* out, FT_UShort flags,
          FT_UShort context_length, int nesting_level )
{
...
  if ( bgc )
  {
    curr_pos = 0;
    s_in      = &in->string[curr_pos];
    bc       = rccsf1->BacktrackCoverage;

    /*Reverse chaining goes from start to end.
    in->pos should be in->length - (in->pos+1)*/
    new_in_pos = in->length - (in->pos + 1);

    for ( i = 0, j = new_in_pos - 1; i < bgc; i++, j-- )
    {
      while ( CHECK_Property( gdef, s_in[j], flags, &property ) )
      {
        if ( error && error != TTO_Err_Not_Covered )
          return error;
        if ( j > curr_pos )
          j--;
        else
          return TTO_Err_Not_Covered;
      }

      error = Coverage_Index( &bc[i], s_in[j], &index );
      if ( error ) return error;
    } }

    curr_pos = in->length - (in->pos+1);
    s_in      = &in->string[curr_pos];
    j = 0;
    error = Coverage_Index( &rccsf1->Coverage, s_in[j], &input_index);

    if ( error ) return error;

    /* starting for lookahead glyphs right after the last context glyph */
    curr_pos += 1;
    if( curr_pos >= in->length ) return TTO_Err_Not_Covered;

    s_in      = &in->string[curr_pos];
```

```

lc      = rccsf1->LookaheadCoverage;

for ( i = 0, j = 0; i < lgc; i++, j++ )
{
    while ( CHECK_Property( gdef, s_in[j], flags, &property ) ) {
        if ( error && error != TTO_Err_Not_Covered )
            return error;
        if ( curr_pos + j < in->length )
            j++;
        else return TTO_Err_Not_Covered;
    }
    error = Coverage_Index( &lc[i], s_in[j], &index );
    if ( error ) return error;
}
ADD_String_ReverseOrder(in, 1, out, 1, val, 0xFFFF, 0xFFFF);
return 0;
}

```

Conclusion

The reverse chain table is useful in modeling the left-to-right context dependency drifts in complex script like Nastaleeq. Implementation of this table has enabled the modeling of Urdu scripts according to its natural way of writing. This code has been integrated into the stable versions of the main Pango libraries available at <http://ftp.gnome.org/pub/GNOME/sources/pango/>. This code may not be exactly similar to the one in the current version of Pango. The code was modified every time a new version of Pango was released.

References

- [1] www.ethnologue.com
- [2] http://en.wikipedia.org/wiki/Aramaic_script
- [3] <http://www.microsoft.com/typography/default.asp>
- [4] Owen, T. "Pango: Internationalized text handling" lwn.net/2001/features/OLS/pdf/pdf/pango.pdf
- [5] Wali, A., Hussain, S. (2006). Context Sensitive Shape-Substitution in Nastaliq Writing System: Analysis and Formulation. International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE2006).