making fonts for the

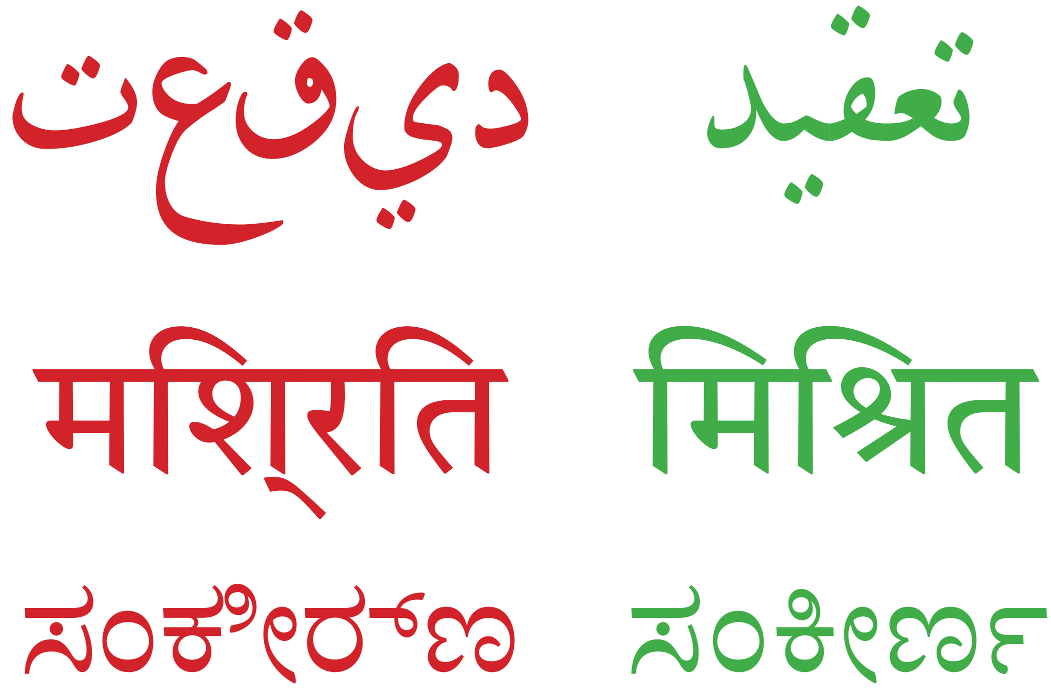
Universal Shaping Engine

John Hudson, Tiro Typeworks Ltd • TYPO Labs, Berlin, 10 May 2016

Version 1.1, 23 May 2016

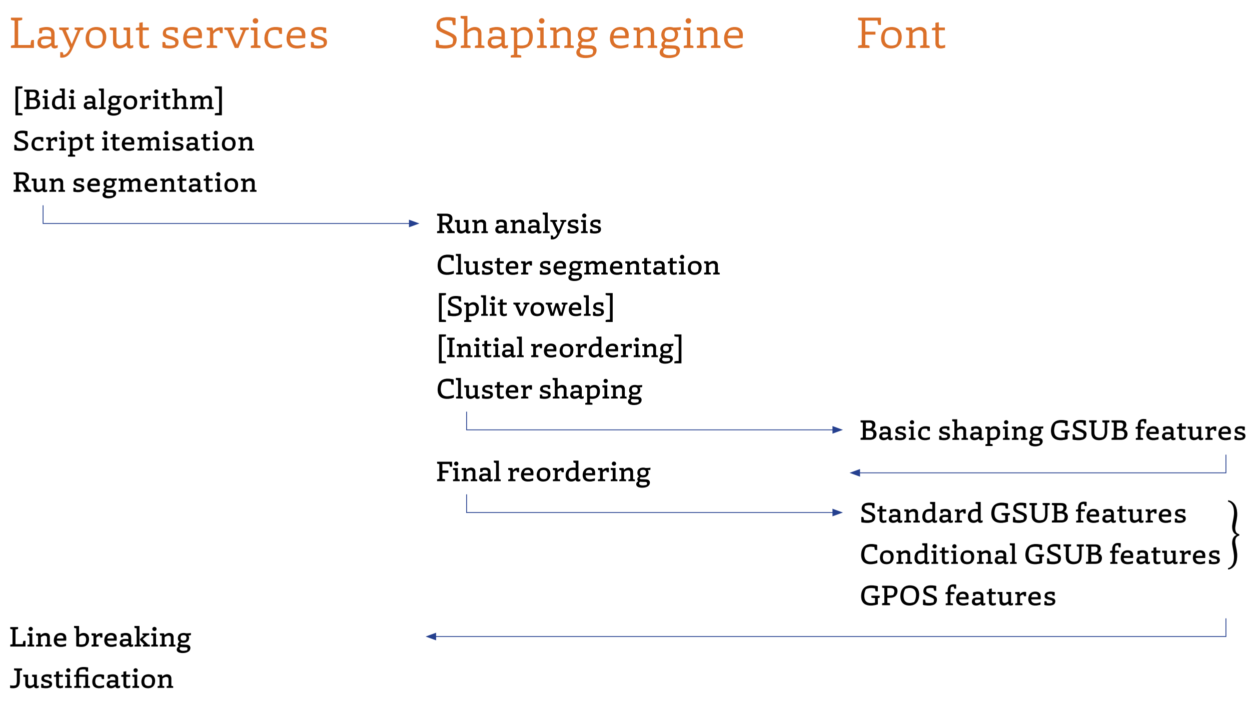
This paper, based on a presentation delivered at the inaugural TYPO Labs font technology conference in Berlin, concerns making a particular kind of OpenType font to work with a new shaping engine for complex script layout. If you're not involved in making fonts for complex scripts, I hope you might still find some interest in the conceptual problems and solutions involved, and also in the insights these provide into the architecture and history of OpenType Layout.

Let me begin by defining what we mean by ‘complex script’. These are scripts that require processing beyond a simple display of the default encoded glyph for each character in order to correctly present text in an acceptably readable form. This processing typically involves character string analysis and manipulation, as well as glyph substitution and positioning. There are, of course, instances in which a font for any script may assume complex behaviours — ligation, contextual substitutions, dynamic mark positioning —, but an inherently complex script is one for which the plain text encoded character sequence will be unreadable without additional processing.



Complex scripts tend to fall into one of two broad categories: those, like Arabic, involving joining behaviour that requires knowledge about adjacent characters and substitution of appropriate forms to display connected lettergroups, and those, like the many Brahmi-derived scripts of South and Southeast Asia, in which the orthographic unit is a cluster that may consist of multiple consonant letters plus dependent vowel sign and additional modifier marks. Scripts in the latter category also tend to involve reordering behaviours, in which there is a distinction between the graphical order of signs in the cluster and their phonetically encoded ordering.

In the OpenType model, complex script layout is handled collaboratively by a shaping engine — residing at the operating system or application level — and the layout in a font. This is a somewhat simplified diagram of that collaboration, and I'm not going to discuss it in a step-by-step way. [For more detailed, see my Unicode conference presentation from 2015.]



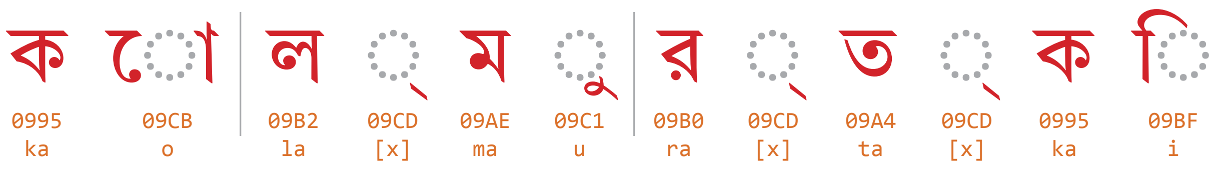
Complex script handling was Microsoft's primary goal in developing a smart font format in the mid-1990s. Microsoft developed Arabic, Hebrew, and Thai shaping engines for TrueType Open, the immediate precursor to OpenType, and in early 1999 shipped the first version of the Unicode Script Processor for Complex Scripts — or Uniscribe — with Internet Explorer 5.01. Subsequent versions of Uniscribe have shipped with all versions of Windows, Office, and Microsoft browsers, often leapfrogging each other in support for additional scripts and languages. Other companies have produced their own OpenType Layout engines for complex scripts, notably the open source Harfbuzz shaper — maintained by Behdad Esfahod —, Adobe's World Ready Composer, and Apple's CoreText engine.

The assignment of scripts to processing by a particular engine generally depends on similarities in shaping needs. This leads to predictable groupings such as the handling of numerous South Asian Brahmi-derived scripts in a common Indic shaping engine, and occasionally to strange-bedfellows, such as assignment of the Thaana script of the Maldive Islands to Uniscribe's Hebrew shaping engine.

The current Windows 10 version of Uniscribe includes nine engines, each of which is responsible for shaping one or more scripts. An engine may also support more than one version of shaping for a given script, mapped to different OpenType script tags, for example the old Windows XP Indic shaping and the new 'Indic2' model introduced in Windows Vista. This enables continued support for older fonts while allowing improved implementations to emerge.

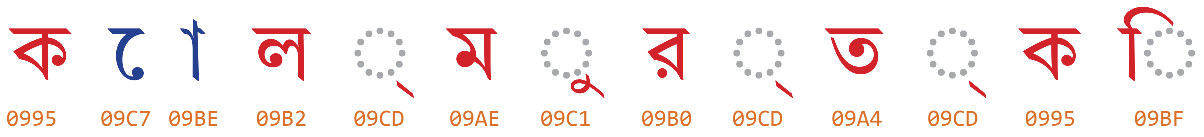
Arabic engine	Arabic, Syriac
Generic engine	Cyrillic, Greek, Latin, etc. (non-complex scripts)
Hangul engine	Hangul, Old Hangul
Hebrew engine	Hebrew, Thaana
Indic engine	Bengali, Devanagari, Gujurati, Gurmukhi, Kannada, etc.
Khmer engine	Khmer
Myanmar engine	Myanmar (Burmese)
Thai/Lao engine	Lao, Thai
Universal engine	Balinese, Batak, Brahmi, Buginese, Buhid, Chakma, Cham, Duployan, Egyptian Hieroglyphs, Grantha, Hanunoo, Javanese, Kaithi, Kayah Li, etc. (45 total)

In case you are unfamiliar with the kinds of things that a shaping engine does with a script, I'll take a moment to discuss a step-by-step example of typical script-specific shaping for a mock character sequence (not a real word). Layout services will have identified this as Bengali, based on the Unicode script property of the characters involved, and will have passed the run to the appropriate Indic shaping engine. The shaping engine has determined that the font supports the Indic2 shaping model using the <bng2> script tag, so is going to apply that shaping model.

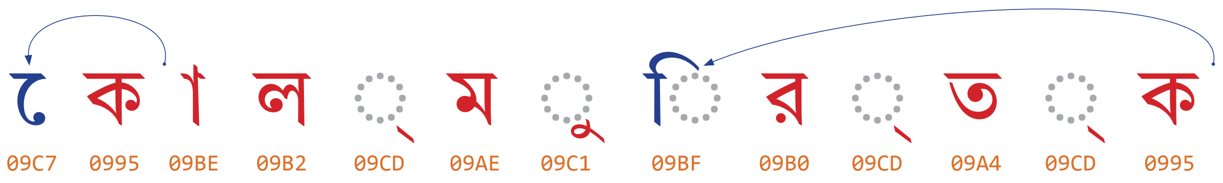


The shaping engine analyses the character run, and segments it into three orthographic units, in this case clusters consisting of one or more consonants with explicit vowel signs. The small diagonal mark (U+09CD) is a vowel killer, indicating that the preceding and following consonants are part of the same cluster.

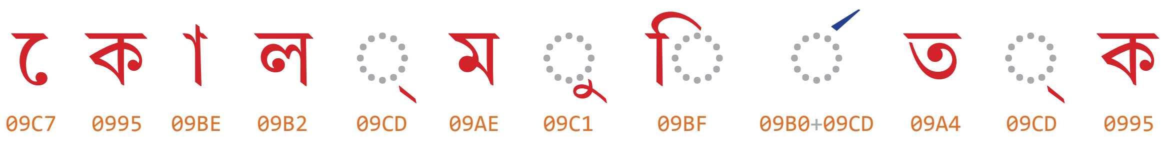
The first step is to split any two-part vowel signs into their constituent elements. This is a buffered character level operation, made possible because both elements are also atomically encoded as characters in Unicode.



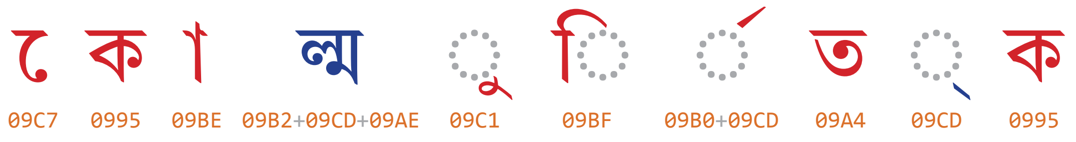
The second step in this shaping model is initial reordering. In our example, this involves moving of left-side vowel signs in the first and third clusters. Again, this is a buffered character level operation: there's no interaction with the font layout tables up to this stage.



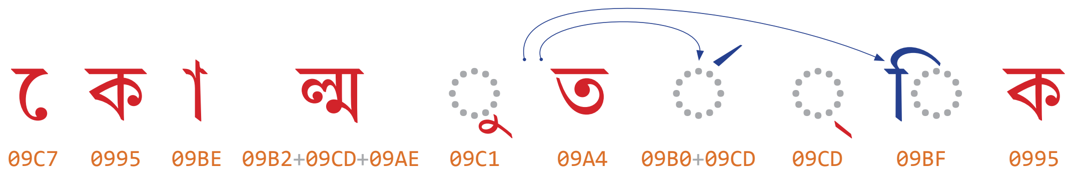
That interaction begins in the third step: application of basic shaping glyph substitution features. These may include precomposition of letter plus nukta forms, and formation of akhand ligatures (a kind of pseudo-letter). In our example, the shaping engine applies the Reph Forms <rphf> feature to the sequence of cluster-initial Ra plus the vowel killer character in the third cluster, substituting the repha mark glyph found in the substitution lookup.



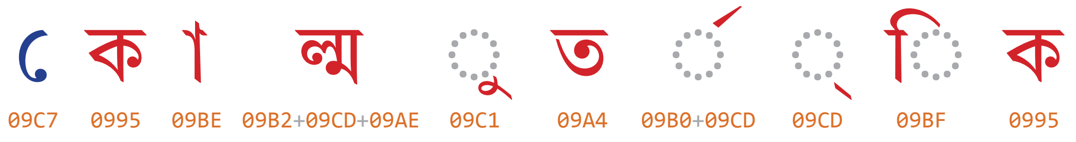
If we had other characters that take special forms in particular situations, these would also be substituted during this phase. Some fonts might substitute half forms of other letter plus vowel killer sequences, although I generally don't do this in Bengali. The next step in our example, is substitution of consonant ligatures in the Conjuct Forms <cjct> feature. Note that in our example, only the conjunct in the second cluster takes a ligature form; this is because the font does not contain a ligature form for the conjunct in the third cluster, which instead will display with an explicit vowel killer sign. This seldom happens in Bengali text, but is the sort of thing that happens when English or other foreign loanwords are transliterated in an Indian script, producing character sequences that don't occur in the local language.



At this stage, basic shaping features are complete, and the next step is final reordering, this time performed at the glyph level, taking output from features such Reph Forms <rphf> and tracking their position in the glyph string. In our example, the left-side ikar (short-i) vowel sign is also re-reordered, following a rule that neither repha nor ikar will relocate past an explicit vowel killer sign, which is an oddity of modern orthographic convention. [If the older convention, which does not employ this rule, is desired, this can be accommodated in how the font lookups are specified and ordered.]



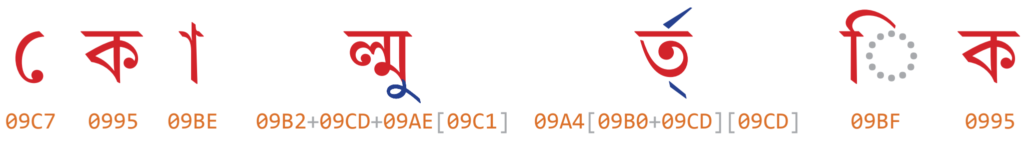
After the reordering, a number of standard substitution features associated with the script are processed simultaneously, i.e. with the order of operations determined by the order of the font lookups in the GSUB table. Depending how the font is made, these may involve additional ligatures, variant forms of marks, mark combinations, etc.. In our example, there is only one such feature to be applied, to substitute the word-initial form of the first vowel sign.



[This is done, in the Indic2 shaping model, with the Initial Forms <init> feature, which is an anomaly not used anywhere else in Indic shaping; indeed, this is the only place in OpenType Layout that requires a shaping engine to be aware of word-level position. In the Universal Shaping Engine, such initial form and similar substitutions would be handled using appropriate contextual lookups.]

Conditional substitutions would also be applied at this stage. These include discretionary typographic features activated by users, but I prefer the term conditional, both because it is less likely to mislead software developers into considering these features optional, and because the conditions to activate the features may be defined by standards for particular kinds of text (for example, ruby notation of kanji characters in Japanese).

Once all substitution features are applied, the shaping engine applies the glyph positioning features. This may involve kerning—none in our example—and, commonly in complex scripts, mark positioning.



The run is now fully shaped, and layout returns to the paragraph level, to line-breaking, hyphenation, and justification. These operations, in theory at least, may involve additional font interactions, but the present reality is that they do not.

কোল্মুর্ত্কি
kolmurtki

So, that's an example of a script-specific shaping engine at work with a compatible font. It works because a lot of knowledge about Bengali character behaviour is baked into the shaping engine, and the font is built to a corresponding specification. This means, of course, that in order for a script to be supported in this way, a shaping engine must be developed for it, and a specification published, and then fonts made and tested with that engine. This can take a long time, which turns out to be only one issue with such an approach.

The cost of intensive development—requiring significant investment of labour, time and money to support a script—, relative to perceived benefit, increases inversely to the diminishment of a script's importance as reckoned in terms of number of users, business case priorities, or geo-political concerns. So scripts with large numbers of users in an identified target market, or that have official status at national or state level, invite intensive development, even if the results end up being only partial in terms of supporting high quality typography. Bengali is spoken by 300 million people, has a relatively high level of literacy—the daily Kolkata newspaper Ananda Bazar Patrika has a circulation of one million copies, and is estimated to be read by five million people every day—, and is an official language of Bangladesh and the Indian the state of West Bengal. It is a language that software companies know they need to support. But there are dozens of South and Southeast Asian scripts, both modern and historical, that do not have official status, are used by minority communities, and may be subject to political pressure to discourage their use. There is very little impetus for software companies to invest in intensive development of shaping engines, specifications, and fonts for such scripts.

Even a script for a major national language with official status, such as Burmese, can fall into a kind of developmental morass, languishing unsupported for more than a decade after inclusion in the Unicode Standard. A number of factors affected this, not least widespread economic sanctions on the military dictatorship in Burma, which prevented North American and European companies doing business in the country prior to the recent democratic reforms. In the meantime, local software developers in Burma produced a non-compatible encoding scheme, hijacking codepoints from the Unicode encoding, which is now causing major headaches for companies belatedly trying to support Burmese in a standard, Unicode conformant way. This has been a 'wake-up call' for many companies: a realisation that the longer you leave a script unsupported, the greater the likelihood that someone will produce a non-standard implementation, accommodation of which may cost significantly more in the long-term.

Script specific shaping engines tend also to be restrictive in terms of the way in which fonts can be built. They may enforce certain conventions about feature ordering and substitution outputs, limiting the freedom of font developers to come up with innovative and possibly more efficient solutions. Such shaping engines tend to reflect a particular software developer's understanding of a script at a particular time, sometimes before anyone has actually tried to build a font to work with engine. If the understanding changes over time, the only way to amend the shaping is to introduce a new script tag and build a new shaping engine—hence the distinction between Indic1 and Indic2 shaping—, with additional intensive development costs.

As well as being unnecessarily restrictive, script-specific shaping engines may be inconsistent, making it difficult for font developers to anticipate how to code OpenType Layout features for a script. If you know how to make a Devanagari font, that knowledge won't necessarily help you to make a Malayalam font, let alone a Khmer or Burmese font. The models employed in the shaping engines are too various, representing the understanding of different developers at different times, each focused on intensively supporting an individual script, rather than on developing a common model.

There are also inconsistencies in interpretation of the script shaping specifications, leading to inconsistencies between implementations. In preparation of the preceding shaping illustrations, for example, I discovered an inconsistency between how Microsoft and Adobe reorder Bengali repha around an explicit vowel killer.

Concentration on the basic shaping needs of complex scripts, without reference to the larger context of typographic communication, has frequently let to partial implementations, to software makers considering the job done when minimally legible presentation of a script is accomplished. It doesn't help that script shaping specifications have tended to discuss only those features necessary to such basic presentation, and have at most glossed over conditional typographic features. When a script is passed to a particular shaping engine for layout, it will likely stay there, and unless that engine makes available all the rich OpenType Layout features that a font maker or user might want to utilise—stylistic sets, variant numeral styles, super- or subscript forms—these will not be available. Many complex script shaping engines can, in this respect, be considered only partial implementations of the scripts they shape.

And then there are the things that are just plain wrong. Microsoft's script shaping specifications focus, for obvious reasons, on run analysis and orthographic unit shaping. This is important, because basic shaping features for Indic scripts have to be applied at the unit shaping level, on individual clusters. Correct shaping of repha and other forms, and of conjunct ligatures, relies on clear definition of cluster boundaries. But after that basic shaping is done, all other features should be applied across the whole glyph run, allowing for cluster interaction in contextual substitutions. Alas, all the script-specific Indic shaping engine makers have interpreted the specifications to mean that all features are applied only at the cluster level, preventing cross-cluster contextual lookups from working and mistakenly applying word-terminal contexts at every cluster boundary.

In summary—and without wanting to diminish the role that script-specific shaping engines have played in enabling people to communicate in complex writing systems—the situation is far from ideal: expensive to develop and maintain, prone to inconsistency and incompleteness, and with bugs that may be impossible to fix without breaking existing fonts and documents.

I gradually became aware of these limitations—as other people have—over more than a decade of making complex script fonts. I doubt if anyone had a better understanding of them than Andrew Glass, who 'owned' the Uniscribe shaping engines as one of his responsibilities at Microsoft. Andrew is one of those people you might not expect to find working in the tech sector, but then you're immediately grateful that he is and see how perfectly qualified his undergraduate studies in Sanskrit and doctorate in Asian Languages and Literature make him for the job. As well as being responsible for Uniscribe from 2009 until recently, he has made fonts for a number of complex scripts, and authored the Unicode encoding proposals for the Brāhmī and Kharoṣṭhī scripts.

When Andrew assumed responsibility for Uniscribe—ten years after the first version shipped—he was asked to create a roadmap for Uniscribe to support all complex scripts in Unicode. Aware of the limitations of the existing model of individual script engines, Andrew had a hunch that a universal model might be possible. It wasn't until 2013, however, that experience building shaping engines for Burmese, N'ko, Javanese, and Buginese, allowed him and his developers to identify the common ground between them and start spec'ing the Universal Shaping Engine.

Andrew was able to make a business case to his managers for the new engine by pointing to the costs of the Burmese encoding fiasco, which had obliged Microsoft and many other companies to accommodate the widely used local hack alongside Unicode. If the time it took to support newly encoded scripts in Unicode could be significantly reduced, there would be far less chance of something like this happening again.

Andrew reasoned that if shaping were driven by standardised complex script character data in Unicode—rather than relying on intensive research and analysis by individual shaping engine makers—, and if that data were passed to a shaping engine with a single, universal cluster model, then each new release of Unicode—now conveniently on an annual cycle—would require only an update of the property data for newly encoded characters. The path from a script encoding proposal to support in software and fonts would be greatly reduced, and a short-term investment in developing the Universal Shaping Engine would replace long-term and increasingly hard-to-justify expenditure on supporting minority scripts and languages on a case-by-case basis.

I'm not sure if Andrew made this additional point to his managers, but relying on character property data in the Unicode standard may also enable companies to side-step geo-political issues associated with some scripts. If support comes simply from updating Unicode data, rather than making a business decision to implement—or not—shaping for a particular script, companies can avoid accusations of either favouritism or neglect.

Let’s now look at how the Universal Shaping Engine works. I’ll speak about each step in more detail, but here is the overview.

Character classification

Two-part vowel splitting

Cluster segmentation and validation

OpenType feature application I

Basic cluster shaping GSUB features

Reordering

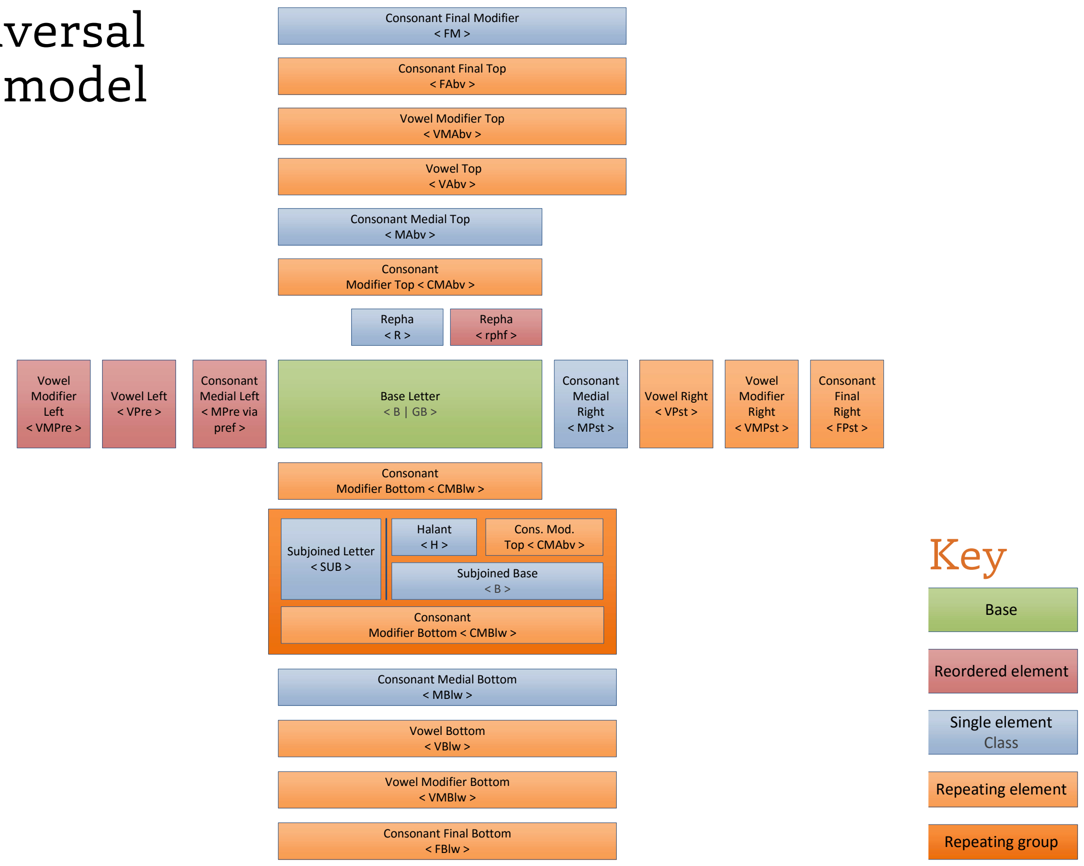
OpenType feature application II

Joining features (`<init>`, `<medi>`, etc.)
Standard GSUB features
Conditional GSUB features
GPOS features

The engine does many of the things that previous, script-specific engines did, but does so in a simplified structure that relies on the font developer making good choices rather than on forcing a particular approach. There is no initial character reordering, but instead there is a new step: character classification based on Unicode properties. Two-part vowel splitting is handled as before, and segmentation of orthographic units or clusters now includes validation against the universal cluster model. Basic cluster shaping features still need to be applied in a distinct phase, and on discrete clusters. After these features are applied reordering happens; this includes both character-level reordering—of left-side vowels, for example—and glyph level reordering of tracked outcomes from two key shaping features: <pref> and <rphf>. After reordering, all OpenType Layout features are applied across the whole glyph run rather than on discrete clusters, and these include both standard features—those presumed to be on by default and either required for correct display of a script or normal to typical typography—and conditional or discretionary typographic features invoked by the user and requested by the client application.

At the heart of the Universal Shaping Engine is the universal cluster model. This is a model of what happens in writing systems at the orthographic unit level: not a model of any particular writing system—indeed, there is no one script that fills all of these boxes—but a superset of behaviours. At the heart of the cluster is a base, around which other graphical elements are arranged, above and below, and to left and right. These elements are ordered relative to the base and to each other. I have used the term element here, rather than character or glyph, because the model is to some extent agnostic about whether the content of the boxes are atomically encoded characters or output from glyph substitution lookups. It is a graphical model, capturing the arrangement of written signs across a multitude of scripts, independent of language or particular orthography, and potentially independent of encoding scheme or font technology.

The universal cluster model


Consonant Final Modifier < FM >
Consonant Final Top < FAbv >
Vowel Modifier Top < VMAbv >
Vowel Top < VAbv >
Consonant Medial Top < MAbv >
Consonant Modifier Top < CMAbv >
Repha < R >
Repha < rphf >
Vowel Modifier Left < VMPre >
Vowel Left < VPre >
Consonant Medial Left < MPre via pref >
Base Letter < B | GB >
Consonant Medial Right < MPst >
Vowel Right < VPst >
Vowel Modifier Right < VMPst >
Consonant Final Right < FPst >
Consonant Modifier Bottom < CMBlw >
Subjoined Letter < SUB >
Halant < H >
Cons. Mod. Top < CMAbv >
Subjoined Base < B >
Consonant Modifier Bottom < CMBlw >
Consonant Medial Bottom < MBlw >
Vowel Bottom < VBlw >
Vowel Modifier Bottom < VMBlw >
Consonant Final Bottom < FBlw >
Key
Base
Reordered element
Single element Class
Repeating element
Repeating group

To get from the specific encoding scheme that Unicode employs for a given script to the universal cluster model requires parsing of Unicode character categories, and translating these into Universal Shaping Engine classes. The Unicode categories are derived from three data sources: the General Category, Indic Syllabic Category, and Indic Positional Category. The initial work on defining the Indic categories, undertaken by Ken Whistler in that late 2000s, provided the basis for Andrew's idea of leveraging Unicode data to shape complex scripts. As you can see in the examples shown in the chart below, there isn't a one-to-one mapping between Unicode categories and Universal Shaping Engine classes, and there are some overrides to Unicode categories built into the engine. The Unicode Indic properties are also still provisional, so we're not quite at the point where shaping can be purely Unicode data driven—'out of the box'—but we're pretty close.

$$\text{Unicode}\left\{\begin{array}{ll}\text{General Category} & \text{UGC}\\ \text{Indic Syllabic Category} & \text{UISC}\\ \text{Indic Positional Category} & \text{UIPC}\end{array}\right.$$

Sigla	USE class	Derivation

B	BASE	UISC = Number;
		UISC = Avagraha & UGC = Lo;
		UISC = Bindu & UGC = Lo;
		UISC = Consonant;
		UISC = Consonant_Final & UGC = Lo;
		UISC = Consonant_Head_Letter;
		UISC = Consonant_Medial & UGC = Lo;
		UISC = Consonant_Placeholder;
		UISC = Consonant_Subjoined & UGC = Lo;
		UISC = Tone_Letter;
		UISC = Vowel & UGC = Lo;
		UISC = Vowel_Dependent & UGC = Lo;

IV	BASE_VOWEL	UISC = Vowel_Independent

IND	BASE_IND	UISC = Consonant_Dead;
		UISC = Modifying_Letter;
		UGC = Po (Punctuation signs)

N	BASE_NUM	UGC = Brahmi_Joining_Number
GB	BASE_OTHER	U+00A0, U+00D7, U+2015, U+2022, U+25CC, U+25FB–25FE
CGJ	CGJ	U+034F

F	CONS_FINAL	UISC = Consonant_Final & UGC != Lo;
		UISC = Consonant_Succeeding_Repha

FM	CONS_FINAL_MOD	UISC = Consonant_Final_Modifier;
		UISC = Syllable_Modifier

M	CONS_MED	UISC = Consonant_Medial & UGC != Lo

…

Sigla	USE subclass	Derivation
FAbv	CONS_FINAL_ABOVE	UIPC = Top
FBlw	CONS_FINAL_BELOW	UIPC = Bottom
FPst	CONS_FINAL_POST	UIPC = Right
MAbv	CONS_MED_ABOVE	UIPC = Top
MBlw	CONS_MED_BELOW	UIPC = Bottom
MPst	CONS_MED_POST	UIPC = Right
MPre	CONS_MED_PRE	UIPC = Left
CMAbv	CONS_MOD_ABOVE	UIPC = Top
CMBlw	CONS_MOD_BELOW	UIPC = Bottom
VAbv	VOWEL_ABOVE	UIPC = Top
	VOWEL_ABOVE_BELOW	UIPC = Top_And_Bottom
	VOWEL_ABOVE_BELOW_POST	UIPC = Top_And_Bottom_And_Right
	VOWEL_ABOVE_POST	UIPC = Top_And_Right

VBlw	VOWEL_BELOW	UIPC = Bottom;
		UIPC = Overstruck

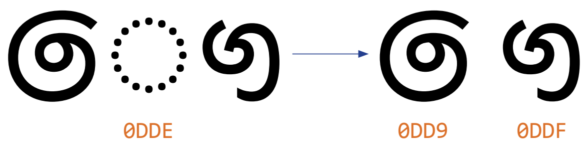
	VOWEL_BELOW_POST	UIPC = Bottom_And_Right
VPst	VOWEL_POST	UIPC = Right
VPre	VOWEL_PRE	UIPC = Left
	VOWEL_PRE_ABOVE	UIPC = Top_And_Left
	VOWEL_PRE_ABOVE_POST	UIPC = Top_And_Left_And_Right

…

This is a good place to start talking about what a font maker needs to think about in order to develop fonts for the Universal Shaping Engine. Obviously, as with any complex script font development, you need to have an understanding of how the script works graphically, and how it is encoded in Unicode. This means doing research, and reading the appropriate sections of the Unicode Standard and any background technical documents such as the original encoding proposals. In addition to this, I would recommend reviewing the Unicode categories for the characters in the script, especially the Indic categories if they exist, and understanding how these will be translated to Universal Shaping Engine classes and hence where they will end up in the universal cluster model.

After character classification, the shaping engine performs split vowel operations. This can happen in two ways: through Unicode canonical decomposition, or via GSUB feature lookups in the font. The first method is obviously simpler, and will be performed whenever a vowel sign character has a canonical decomposition to two separate characters as in the Sinhala example shown here.

Unicode decomposition based



```
0DDE;SINHALA VOWEL SIGN KOMBUVA HAA GAYANUKITTA;Mc;0;L;0DD9 0DDF;;;;N;;;;;
```

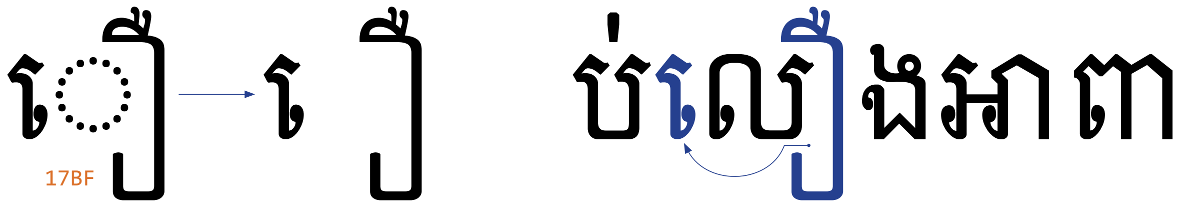
Most two-part vowel signs in Unicode, to-date, have decompositions, either because the component elements are identical in form and behaviour to stand-alone characters, or because the encoding model provides for such decomposition with atomic encoding of split-vowel parts. There are, however, examples of two-part vowel sign characters without canonical decomposition, notably in the Khmer encoding, and this approach could be followed for future script additions to Unicode.

If, in future, you're faced with such characters when developing a font for the Universal Shaping Engine, you will need to split the vowel at the glyph level using a one-to-many GSUB lookup (type 2). Thanks to the character classification stage, the Universal Shaping Engine will already be tracking the two-part vowel characters as it processes the basic shaping features. This is particularly important in the case of split vowels in which one of the elements will take a pre-base position in reordering. This is illustrated here using a hypothetical Khmer implementation for the Universal Shaping Engine (in Uniscribe and elsewhere, Khmer already has its own script-specific engine, which works somewhat differently from this).

In this example, character classification has already identified the Khmer vowel sign ya as a two-part vowel in a class that includes PRE. The engine looks for the default glyph for that character in the basic shaping features, and when it sees a lookup that maps to two new glyph IDs—hypothetically in the Post-Base Forms <pstf> feature, although a dedicated feature may be defined for this purpose—, it tracks the first of these glyphs and at the reordering stage will move it to the appropriate position in the cluster model.

GSUB type 2 based

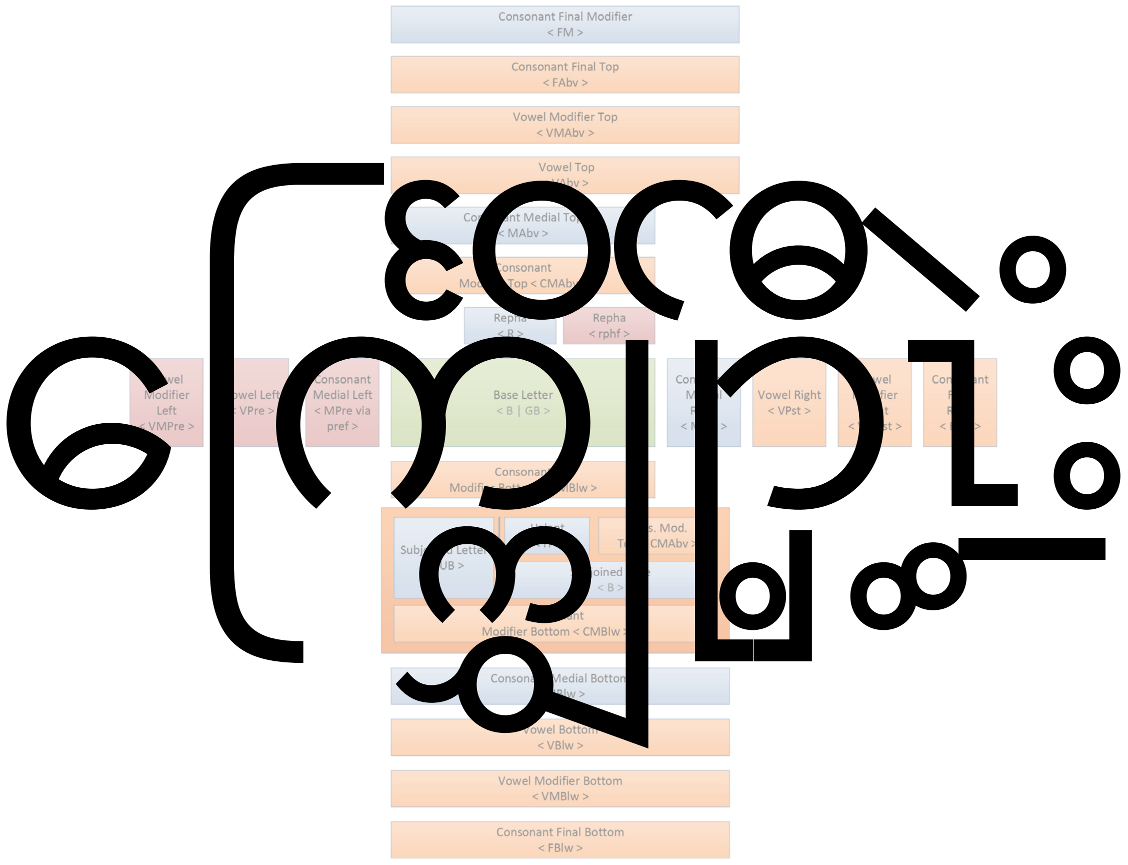
```
USE subclass : VOWEL_PRE_POST     Derivation : UIPC = Left_And_Right
```



```
<pstf> : khSignYa -> khYaPRE khYaPOST
```

Font developers don’t need to know much about cluster validation, other than to understand what will constitute separate clusters for the script. At this stage the engine checks that the encoded character string is well-formed and conforms to the universal cluster model. Since the model is generalised and permissive, accommodating the graphical features of many different scripts, the validation ignores particular orthographic or linguistic rules, leaving these to spell-checkers.

Because no assumptions are made based on linguistically permissible sequences, it is possible to test maximal cluster shaping for a given script. In the foreground is a maximal Burmese cluster, containing a base character plus all the different characters, fully shaped, that may surround it in a cluster. Of course, all these characters never occur together in actual Burmese text, but for cluster validation purposes this is a well-formed character sequence.



Consonant Final Modifier < FM >
Consonant Final Top < FAbv >
Vowel Modifier Top < VMAbv >
Vowel Top
Repha < R >
Repha < rphf >
Consonant Medial Left < MPre via pref >
Base Letter < B | GB >
Vowel Right < VPst >
Consonant Modifier Bottom < CMBlw >
Vowel Bottom < VBlw >
Vowel Modifier Bottom < VMBlw >
Consonant Final Bottom < FBlw >

After validating the cluster, the Universal Shaping Engine applies specific OpenType Layout features needed to resolve basic shaping forms and prepare the string for reordering. These features are processed in three groups, and lookups for these features should be arranged in a corresponding order.

Default glyph pre-processing features

`<locl>`	Localized Forms
`<ccmp>`	Glyph Composition/Decomposition
`<nukt>`	Nukta Forms
`<akhn>`	Akhands

Reordering features

`<rphf>`	Reph Forms	Output reorders after the base
`<pref>`	Pre-Base Forms	Output reorders before the base

Orthographic unit shaping features

`<rkrf>`	Rakar Forms
`<abvf>`	Above-Base Forms
`<blwf>`	Below-Base Forms
`<half>`	Half Forms
`<pstf>`	Post-Base Forms
`<vatu>`	Vattu Variants
`<cjct>`	Conjunct Forms

The first group of features performs pre-processing substitutions of default glyphs, including localised forms, and composition or decomposition for downstream processing, for example combining Indic letters plus nukta dots into precomposed glyphs, or forming 'akhand' ligatures for pseudo-letters. The ordering of processing within the group is determined by the lookup order in the font.

I'll talk about the reordering feature group in more detail, with examples, in the next section. These play an important role in the Universal Shaping Engine, which tracks the output of these two features for reordering purposes. These features are processed individually, in this order: Reph Forms first, and then Pre-Base Forms.

The orthographic unit shaping features are not applied in a fixed order, so lookup ordering can be important, and will depend how you plan to use the features for shaping a particular script. Not all scripts or fonts will require all of the features in any of the basic shaping feature groups.

The most important things for font makers to bear in mind when implementing features for this stage are:

a) that no reordering has taken place at this stage, so the order of glyphs in the string and, hence, in lookup inputs, is the order derived from the Unicode character sequence, as mapped through the cmap table to default glyphs;

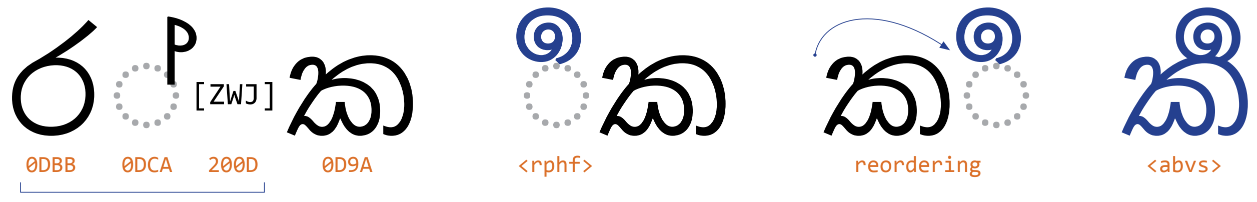
b) that the pre-processing and orthographic unit shaping feature groups are applied according to their respective lookup ordering in the font, controlled by the font maker, while the reordering features are discretely processed; and

c) that output from the Reph Forms feature will be reordered after the base character in the cluster, while output from the Pre-Base Forms feature will be reordered before the base character.

Let's look a bit more closely at the two reordering features. The Reph Forms feature was originally registered to handle, specifically the conjunct-initial ra in Indic scripts, which typically reorders after the base and other consonants, either as a mark or as a stand-alone sign. Since it is encoded, in phonetic order, at the beginning of the cluster, it needs to be reordered when it takes a repha form. The Reph Forms feature is applied by the shaping engine on the first three characters in the cluster, which captures both those scripts in which repha is formed from ra plus the vowel killer and those such as Sinhala, shown here, where the Zero-Width Joiner control character is also required.

In the Sinhala example, the Reph Forms feature lookup substitutes the repha mark for the input glyph string, and the Universal Shaping Engine tracks the latter, so that at the reordering stage it can reorder that glyph after the base, to the appropriate place in the cluster model. In this font, the combination of the base letter plus repha is resolved, after reordering, with a ligature in the Above-Base Substitutions feature during the second OpenType Layout stage; of course, a mark-to-base GPOS anchor attachment method could be used instead.

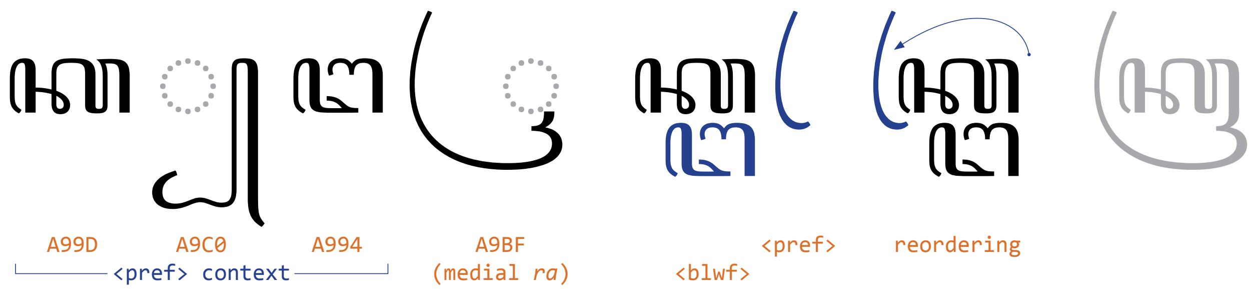
<rphf> Reph Forms



In this example, the Reph Forms feature is being used to substitute an actual repha form, but in the Universal Shaping Engine it can be used generically to identify to the engine any element that has the reordering properties of a repha. The engine does not require the input for this feature to be ra plus vowel killer; it does, however, require that this feature produce only a single, trackable glyph per cluster.

In the same way that the Reph Forms feature can be used to move any element with repha reordering properties from before the base to after the base, the Pre-Base Forms feature can be used to move an element from after the base to before it: specifically, to the position in the cluster model occupied by a prescript medial consonant. This illustration shows the shaping of such a medial consonant in the Javanese script.

<pref> Pre-Base Forms



This is a particularly interesting example, because the Pre-Base Forms lookup is contextual. As shown in the grey cluster on the right, the medial ra character normally joins to a preceding base and takes this sweeping form under the base and rising before it. In order to attach to the base in this way, it needs to come after the base in the glyph string. But when the base is followed by an intervening vowel killer plus second consonant that takes a below-base form, as in the example, the medial ra is represented by a stub form on the left of the cluster. So the <pref> feature substitutes this left-side form, based on the context of the preceding letter and vowel killer sequence. That sequence is going to be resolved to a subscript letter, in the Below-Base Forms <blwf> feature in the orthographic unit shaping feature stage, but because the Pre-Base Forms feature is applied discretely, before that stage, the context string has to be the complete sequence with the vowel killer and the default consonant forms, not the subscript glyph. It is always important, when coding the lookups, to keep track in your mind of what the current state of the glyph string will be at the moment the lookup is applied by the engine. As with the Reph Forms feature, the Pre-Base Forms feature must only produce a single, trackable glyph per cluster.

After all basic shaping GSUB features are applied, the string is reordered by the engine. As just discussed, feature-based reordering is effectively driven by the font lookups in the two reordering features, with some restrictions on their output.

Property-based reordering is derived, ultimately, from Unicode Indic categories, but at this stage is based on Universal Shaping Engine classes. Note that these may derive from the Unicode categories during the character classification stage, or may be inferred from output from two-part vowel splitting. Because reordering takes place after that vowel splitting, it affects only the pre-base vowel parts.

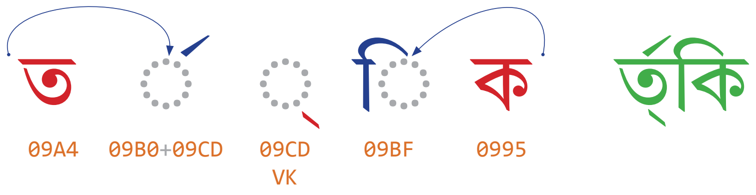
Feature-based reordering

rphf	Reph Form	Moves after base	Repha or similar
pref	Pre-Base Form	Moves before base	Pre-base medial consonant or similar

Property-based reordering

R	REPHA	Moves after base	Atomically encoded repha
VPre	VOWEL_PRE	Moves before base	Pre-base vowel, including pre-base element from 2-part vowel splitting
VMPre	VOWEL_MOD_PRE	Moves before base	Pre-base vowel modifier, moves before pre-base vowel

Following the modern orthographic convention used in India, repha and pre-base vowel signs will not reorder past an explicit vowel killer sign. In other words, if a vowel killer character remains in the middle of a cluster after the basic shaping features have been applied, this acts as a kind of roadblock to the reordering move. This is illustrated if we imagine the Bengali cluster example I used earlier being shaped by the Universal Shaping Engine. If you want to avoid this result, then you need to ensure that your basic shaping form features and glyph substitutions are formed in such a way that no vowel killer remains in the string when reordering is applied.



After reordering, the second phase of OpenType Layout features is applied, beginning with the joining features: Initial Forms, Medial Forms, etc.. To this point in the presentation, I have been mostly talking about complex script handling in terms of alphasyllabic scripts of South and Southeast Asia, but the Universal Shaping Engine is designed to also be able to handle Arabic-like scripts with letterform joining behaviour based on Unicode joining categories. Indeed, it should be able to handle a script that combines Indic- and Arabic-like behaviours, if such a thing existed and were encoded in Unicode with appropriate character properties and categories.

Joining features

`<init> <medi> <fina> <isol>`

Standard GSUB features

`<abvs> <blws> <calt> <clig> <haln> <liga> <pres> <psts>`
`<rclt> <rlig> <vert> <vrt2>`

Conditional GSUB features

All the rest!

GPOS features

`<curs> <dist> <kern> <mark> <abvm> <blwm> <mkmk>`

I have listed the standard and conditional GSUB features separately, but these will actually be processed simultaneously by the engine. That is to say, the processing sequence and interaction of all remaining glyph substitution is based on the order of the lookups in the font. Further, all these features are applied across the whole glyph run, rather than at the cluster level, allowing cross-cluster contextual lookups and rich interaction between the shaped clusters. So long as you have a good handle on what glyphs will be in the string after basic shaping, and in what order they will be after reordering, you—as the font maker—have great freedom and creative flexibility in this stage.

GPOS features are also applied simultaneously, allowing similar freedom, although the order shown here is recommended. Think of it in terms of moving outward from joined spacing glyphs attached with cursive connection lookups, to adjusted spacing of glyphs using kerning or similar lookups, to zero-width marks attached to spacing glyphs, and finally marks attached to other marks.

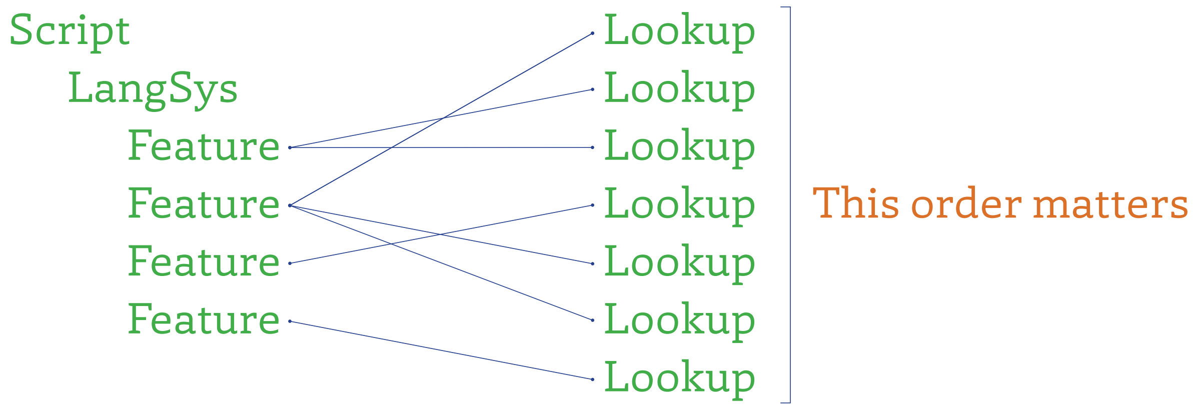
[I have decided not to take time in this paper to discuss mark filtering, which is a common technique used in complex script lookups. I presume that most OpenType font makers have a reasonable idea of how this works, and what the implications are for GPOS.]

I want to talk briefly about lookup ordering, because I worry that the limitations imposed in older shaping engines, in partial OpenType Layout implementations, and even in font tools have tended to obscure the original model. There is a tendency to think about OpenType Layout in terms of a hierarchical tree of script, language system, features, and lookups, and hence to think about processing order as being primarily associated with features.

- Script
  - LangSys
    - Feature
      - Lookup
      - Lookup
    - Feature
      - Lookup
      - Lookup
      - Lookup
    - Feature
      - Lookup
    - Feature
      - Lookup

Complex script shaping engines have tended to reinforce this incorrect impression by discretely processing some features in a fixed order.

The actual structure looks more like this: a hierarchical tree of script, language system, and features independent of a block of lookups. It is a model that should provide a lot of flexibility to font makers, allowing, among other things, interleaving of lookups associated with different features. [One of the reasons why I still favour Microsoft's VOLT tool is that it makes this structure explicit in its UI.]



The original intent was that layout engines would act as an interface between applications and fonts, activating features for text as appropriate to the script, language system and feature state—required, standard, conditional—but would allow the order of lookups in the font to determine how those features were processed and how they would interact. Two things happened in the early development of complex script support to undermine this model. The first was that layout engine developers realised that in order to perform complex script shaping reliably, they needed some processing to happen in a particular order, and to break the processing into multiple stages around reordering. The second was that some font makers got that ordering wrong in the lookups. Rather than sticking to the model and requiring font developers to fix the fonts, Microsoft opted to enforce feature-based processing order in shaping engines, bypassing the lookup ordering for some features. This resulted in a complex script font development environment that is both more restricted than originally intended, and also more confusing: it isn't always clear which features a shaping engine will process in a fixed order and which will be processed according to lookup order.

One of the goals of the Universal Shaping Engine is to limit as much as possible the application of fixed feature processing, and to return to the font maker a large amount of the original freedom of the OpenType model. Of course, freedom means responsibility: a lot more of the control of what happens in complex script display now resides in the font, so it is important to understand the stages that the Universal Shaping Engine employs, and to order lookups accordingly.

It is difficult to talk about or diagram lookup ordering independent of a particular font, since different fonts may have very different lookup sets. So we're obliged to talk instead about ordering of 'lookups associated with features', using the features as a kind of key to the lookups. Obviously, this doesn't expose possible interleaving of lookups associated with different features within the processing blocks.

Here is a cheat sheet for lookup ordering for the Universal Shaping Engine processing stages. As you can see, there are only two features that have fixed ordering, and the processing of the great majority of features is determined by lookup ordering.

USE lookup ordering cheat sheet

`locl,ccmp,nukt,akhn`	Free order; this recommended
`rphf,pref`	Fixed order
`rkrf,abvf,blwf,half,pstf,vatu,cjct`	Free order; this recommended
`isol,init,medi,fina`	Free order; this recommended
`abvs,afrc,blws,calt,case,clig,cpsp, cswh,cv01-cv99,c2pc,c2sc,dlig,dnom, falt,frac,haln,hist,hlig,jalt,liga, lnum,ltra,ltrm,mgrk,nalt,numr,onum, ordn,ornm,pcap,pnum,pres,psts,rand, rclt,rlig,rtla,rtlm,salt,sinf,smcp, ss01-ss20,stch,subs,sups,swsh,titl, tnum,unic,vert,vrt2,zero`	Free order *CJK- and math-specific features not listed*
`curs,dist,kern,vkrn,mark,abvm,blwm, mkmk`	Free order; this recommended *CJK-specific features not listed*

The Universal Shaping Engine was invented at Microsoft, but Andrew and his colleagues—including Peter Constable, who has taken over responsibility for script shaping—have actively encouraged other companies to implement the engine in their own software, and to ensure compatibility. The specification is published—although in need of some updates and clarifications—, and the engine was the subject of several days of discussion at the first meeting of the ad hoc OpenType Working Group, hosted by Google in 2014. In addition to shipping in the Windows 10 version of Uniscribe, the Universal Shaping Engine has been implemented in the open source HarfBuzz text shaper, and I am aware of one other implementation that I am not allowed to talk about yet, but which I expect to be announced soon.

It's called the Universal Shaping Engine, so it's reasonable to ask how universal it actually is. The answer is not really and not quite. It isn't really universal in terms of the scripts that it currently shapes: at present, only a subset of complex scripts is passed to the Universal Shaping Engine. The list shown here are those for which Uniscribe currently uses the engine (Harfbuzz uses it for a different set; yes, this inconsistency is of concern). These are all either previously unsupported, newly encoded, or—in the case of Sinhala and Tibetan—scripts that previously had their own script-specific implementations but which Microsoft have migrated to the new engine (also of concern, as my understanding is that Microsoft tested only their own Sinhala and Tibetan fonts before making this change). Many of these scripts are minority writing systems, some of them not in modern use.

USE shaped scripts in Windows 10

Balinese
Batak
Brahmi
Buginese
Buhid
Chakma
Cham
Duployan
Egyptian Hieroglyphs
Grantha
Hanunoo
Javanese
Kaithi
Kayah Li
Kharoshthi
Khojki
Khudawadi
Lepcha
Limbu
Mahajani
Mandaic
Manichaean
Meitei Mayek
Modi
Mongolian
N'Ko
Pahawh Hmong
Phags-pa
Psalter Pahlavi
Rejang
Saurashtra
Sharada
Siddham
Sinhala
Sundanese
Syloti Nagri
Tagalog
Tagbanwa
Tai Le
Tai Tham
Tai Viet
Takri
Tibetan
Tifinagh
Tirhuta

Most of the previously supported major complex scripts are still being passed to their older engines. There are, however, advantages to efficiency in passing Indic scripts to the Universal Shaping Engine—as well as all the benefits to font makers discussed in this presentation—, so it seems increasingly likely that new Indic script tags such as <dev3> will be defined in order to be passed to the new engine. [It's too dangerous simply to start passing the existing Indic2 script tag to the Universal Shaping Engine, because too many fonts have been made in ways that might break if one starts, for example, suddenly applying standard features across cluster boundaries.]

It's called the Universal Shaping Engine, then, not because it shapes all scripts, but because it uses a universal model. Of course, as soon as you declare that you have a universal model, someone comes along with an exception to that model. In this case, the exception is the Tai Tham or Lanna script of northern Thailand, which uses subjoined consonants in ways that may compress multiple syllables into a single cluster, causing recursion in cluster analysis. It remains to be seen whether Tai Tham can be accommodated with exception code in the Universal Shaping Engine, or will need to be passed to a script-specific engine.

So the Universal Shaping Engine turns out to be not quite universal in this sense either. But it's pretty darn close, and is a remarkable achievement.

In summary, this is my advice for type designers and font technicians making fonts for the Universal Shaping Engine:

- Know your script: do research; understand how the writing system functions as a system;
- Know how Unicode encodes your script; be aware of the possible role of control characters, such as the Zero-Width Joiner and Non-Joiner, which might need to be included in GSUB lookups;
- Know how the Universal Shaping Engine works: review the specification and watch for updates; review this presentation (which I'll publish online in a couple of weeks); ask questions;
- Plan your glyph set carefully, considering in advance everything you might need, how it will be processed by the shaping engine, and how you will handle the layout in GSUB and GPOS; consider making a spreadsheet to manage the project;
- Check your workflow and toolset: make sure the font creation software you are using lets you do everything you want to do in terms of feature coding and lookup ordering;
- Always remain aware of the order of glyphs in the cluster at different stages of processing, and hence the input and output of sequential lookups;
- Enjoy the freedom and flexibility that the Universal Shaping Engine model provides, but be ready to embrace what that means in terms of responsibility for making the font work.

Select links and acknowledgements

microsoft.com/typography/OpenTypeDev/USE/intro.htm
Universal Shaping Engine specification (watch for updates and fixes)

blogs.windows.com/windowsexperience/2015/02/23/windows-shapes-the-worlds-languages/
Introductory article on the Universal Shaping Engine in Windows 10

tiro.com/John/Hudson-Soyombo-DECK.pdf
Soyombo script test case for the Universal Shaping Engine (Unicode conference 2014)

tiro.com/John/Hudson_IUC39_Beyond_Shaping.pdf
General overview and critique of OTL implementation (Unicode conference 2015)

Many thanks, for their contributions and assistance, to Andrew Glass & Peter Constable (Microsoft); Behdad Esfahod & Roozbeh Pournader (Google); Martin Hosken (SIL); Anshuman Pandey; Ben Mitchell.

With gratitude to Jürgen Siebert and Jörg Schweinsberg at Monotype for the invitation to speak at TYPO Labs, and to the conference organisers and staff, especially Franziska Parschau.