ENHANCING CONTEXTUAL SUBSTITUTION SUPPORT IN PANGO USING OPENTYPE

MS Thesis for the Degree of

Submitted in Partial Fulfillment
of the Requirements for the
Degree of

Master of Science (Computer Science)

at the

National University of Computer & Emerging Sciences

By

Aamir Wali

September 2004

Approved:

____
Dr. Fakhar Lodhi
Head of the Department of Computer Science

____

Supervisory Committee

Advisor:

____

Mr. Shafiq ur Rehman
Associate Professor
Department of Computer Science
NUCES Lahore

Co –Advisor

____

Dr. Sarmad Hussain
Associate Professor
Department of Computer Science
NUCES Lahore

Vita

Mr. Aamir Wali was born in Lahore, Pakistan on September 04, 1979. He received a Bachelor of Science in Computer Science from National University of Computer and Emerging Sciences, Lahore in 2003. His area of interest includes script processing in general and Font development and Typography in particular.

Abstract

This thesis proposes better support for context sensitive substitution in Linux. The support is made available through Pango. Pango utilizes the OpenType formulism. The incorporation of contextual substitution support in Pango was directed towards two dimensions. The first dimension focused towards the completeness or on the physical expansion of substitution support in Pango library. This includes the inclusion and fixing of the OpenType tables that actually provide contextual substitution. The other dimension is performance. It was observed that the OpenType rule processing algorithm was very inefficient. A more efficient mechanism for processing the lookups on the input string is proposed. The results showed that the processing time with the new algorithm was reduced to 20% of the original time.

Acknowledgements

I am very thankful to Mr. Shafiq-ur-Rehman for his help and guidance during the thesis. His timely suggestions helped me in completing the thesis in the best possible manner. I am also grateful to Dr. Sarmad Hussain for his suggestions and encouragement throughout my graduate studies. Special Thanks to our calligrapher, Mr. Jamil-ur-Rehman who provided insight into Nastaleeq Script and patiently listened to my long unresolved theories.

I would like to thank all fellow graduate students who have supported and assisted me throughout my graduate years in NUCES. Moreover, thanks to my best friend Atif who made Linux operational on VMWare. I would also like to thank the undergraduate students especially Noman Mushtaq and Nadir who helped me in the Linux Installation process.

And finally, I should like to thank my parents for their full support and encouragement during my studies.

Table of Contents

List of Figures

List of Tables

PART ONE

The Mainstream

Chapter 1

Introduction

Urdu is the National language of Pakistan. People of Pakistan are contributing a lot in the field of Information technology, yet the full potential is not realized. That is mainly because most of the masses are still not using the computer and the main reason for this is of the language. To teach English language to a huge population (140 million) is simply not possible. On the other hand, an arrangement is required that could make possible for the masses to use the computer in their own language.

As the need for multi-lingual support in computer system grew, the convention TrueType technology was found to be inadequate. It utilizes a one to one correspondence between a character and the glyph. This would definitely not work for Urdu because the writing systems that are traditionally used for writing this language are far too complex to be processed by the TrueType specification. Urdu language exhibits both positional substitution i.e. a letter has four forms and contextual substitution in which a letter in one given form may have different shapes depending on following (and/or preceding) characters. Therefore to enable Urdu, complex script processing is required that can handle this two-dimensional context sensitive behavior of Urdu

The recently developed OpenType technology enables advanced text layout by embedding the complex script processing logic within the font. OpenType specification is designed to be fully capable of supporting Urdu and in fact has already been successfully incorporated in Windows. Meanwhile such supported has not really matured in Macintosh or Linux/Unix although these platforms are fully capable of doing so. Macintosh is based on its own AAT i.e. Advanced Apple Typography model quite different from the OpenType. Linux however complies with the OpenType and can well be extended to provide better Urdu support than it currently does.

This thesis proposes a complete and efficient contextual substitution support in Linux through Pango using OpenType. This will involve first the inclusion and fixing of the OpenType tables that actually provide support for contextual substitution. This will then be extended to improving the performance of the lookup processing algorithm that processes the information in these tables (or rules) on the input string.

The succeeding chapters of this dissertation are organized as follows: Chapters 2 gives the background of advanced text processing in Windows. Chapter 3 discusses the level of support in Linux. Chapter 4 describes the Pango text processing library and the application of this library on Arabic text in chapter 5. Chapter 6 and 7 discusses the some of the problem regarding OpenType support and processing in Linux. Chapter 8 lists the concluding research problem for this dissertation.

Chapter 8, 9 and 10 describes some of the proposed solutions to the problem. The results are given in chapter 11. Chapter 12 concludes the document and gives some insight into future work.

Chapter 2

Background

The earliest font technology that existed traces back to 'bitmap fonts'. In these fonts each character shape was stores a bitmap consisting of series of pixels. Bitmap fonts then created characters by arranging bits (or pixels) in specific patterns. Because there was no good way to extrapolate between one font size and another, several bitmaps fonts had to be developed (or purchased) for each point size. The other disadvantage of bitmap fonts is that for large point sizes the displayed font appeared jagged. Increase in font size was yet another drawback.

Outline fonts soon overtook Bitmap fonts. Outline fonts use (d) mathematical description of characters that represent each character as an outline consisting of series of contours well defined by points. These mathematical descriptions can easily be scaled up or down to a wide range of sizes by simple multiplication by a scaling factor. Only one fonts had to be developed (or purchased) and a wide variety of computations such as rotating, slanting to filling can be applied.

Characters in bitmap fonts can be output to the screen or printer in the same form they are stored. Outline fonts however must go through some additional steps. They are first scaled, then outlined and finally converted to bitmap.

The outline font technology led to the designing of various outline-font standards. Two such standards that are also currently available are TrueType and Type 1 font formats.

2.1. Type 1 & TrueType fonts

The Type 1 font format was developed by Adobe. Apple Computer Inc originally designed the TrueType outline font standard. The primary difference in the two standards was the form in which the outlines were stored i.e. how they describe a letter's shape by means of points, which in turn define lines and curve, was different. Type 1 stores glyph as outlines represented by third other Bezier splines while TrueType store them as outlines indicated by second order b splines.

TrueType font was developed in 1980, six years after Type1. Apple initially code-named it as Royal and later introduced it as TrueType. At that time, it was considered a means of a better outline font with good hinting capabilities and a solution to some of the technical limitations of Adobe's Type 1 format [10].

The TrueType format was designed to be efficient in storage and processing, and extensible. It was also built to allow the use of hinting approaches already in use in the font industry as well as the development of new hinting techniques, enabling the easy conversion of already existing fonts to the TrueType format. This degree of flexibility in TrueType's implementation of hinting makes it extremely powerful when designing characters for display on the screen. Microsoft had also been looking for an outline format to solve similar problems, and Apple agreed to license TrueType to Microsoft.

Apple included full TrueType support in its Macintosh operating system, System 7, in May 1990. TrueType specifications were made public and Microsoft first included TrueType in Windows 3.1, in April 1991. The fonts developed were based on TrueType standards; hence the name TrueType fonts. Soon afterwards, Microsoft began rewriting the TrueType rasterizer to improve its efficiency and performance and remove some bugs (while maintaining compatibility with the earlier version). [1]

2.2. Unicode

The TrueType fonts implementing the TrueType standard were initially based on 7-bit ASCII. This means that they did not accept any input beyond the 7-bit range (0-127). With the idea of multi-lingual text processing this problem became apparent and a question arose that *how can letters of all languages be represented on a computer?*

Well, the ASCII's 7-bit character size was inadequate to handle multilingual text, so a Consortium by the name of Unicode adopted a 16-bit architecture, which extends the benefits of ASCII to multilingual text. Unicode characters are consistently 16 bits wide, regardless of language, so no escape sequence or control code is required to specify any character in any language. Unicode character encoding treats symbols, alphabetic characters, and ideographic characters identically, so that they can be used simultaneously and with equal facility. Computer programs that use Unicode character encoding to represent characters but do not display or print text can (for the most part) remain unaltered when new scripts or characters are introduced

Unicode is a 16-bit character-encoding standard that represents most of the characters used in general text interchange throughout the world. Unlike other character encoding standards that assign character codes to both characters and glyphs, Unicode assigns character codes only to characters. In Unicode, each character has a distinct linguistic function or meaning, and its character code is unambiguous. Character codes are not assigned to glyphs or glyph variants because a glyph is simply a graphic depiction that has no meaning apart from the character or characters that it represents. By functionally separating characters and glyphs, Unicode simplifies text processing for software developers and users.

Unicode solved the problem how characters belonging to all the languages are to be stored. However, a one letter multiple shape problem remained. The TrueType font allows a one to one correspondence between characters in a coded-character set e.g. ASCII or Unicode and the glyph in the font that represents the character. This model does not work well for languages that require complex script processing. For example in Urdu a letter can have 4 different shapes. These are position dependent. Consider the following table 1 in which letter bay indicated in gray has a different shape when it occurs in a) initial, b) medial, c) final and d) isolated position.

با	قبا	قب	ب
(a)	(b)	(c)	(d)

Table 1 Different forms for Urdu letter ***bay***

This was solved by assigning an additional Unicode to each of initial, medial and final form whereas isolated already had the ‘default’ Unicode. So these forms, which came to be known as presentation forms [9], were included to provide users with a simple method to generate them But the Nastaliq font consists of hundred of different shapes and it would be infeasible to assign a Unicode to each of its presentation for.

So it was felt that there is no need to include any additional presentation shapes in Unicode, in fact all the various shapes and ligatures of the Arabic script should automatically be generated by the rendering engines of the software that implements Unicode. Efforts were made to remove the language barrier. Different technologies were developed and different paths adopted. The next section gives a detail description of such efforts.

2.3. OpenType

OpenType is a font format developed and promoted jointly by Microsoft and Adobe. As with AAT, it starts with ‘sfnt’ data structure as the basis; it uses fonts that adhere to the basic directory-table structure introduced with TrueType. The natural solution for the Windows platform is to consider Microsoft’s solution to the smart font rendering problem viz. OpenType. 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 OpenType font format supports international typography with five new tables: GSUB, GPOS, BASE, JSTF, and GDEF. TrueType Open provides typographic and linguistic information for properly positioning and substituting glyphs, operations that are required for accurate typographic composition in many language environments. With OpenType, a single font may support multiple scripts. To assist application and OS text processing clients, the OpenType data is organized by script, language system, and typographic feature [7].

Script > Language System > Feature > Lookup

The GSUB and GPOS tables define glyph substitution and positioning features. GSUB supplies five types of substitutions to support different kinds of character-to-glyph mappings, such as many-to-one ligature glyph substitution and one-to-many ligature decomposition. GPOS defines seven types of positioning features that provide two-dimensional positioning data for adjusting glyph placement and for glyph attachment.

The BASE table contains baseline and minimum/maximum extent data for each script. Script baselines can be defined in relation to one another to properly align glyphs from different scripts. Min/max extent values can be modified for particular language systems or features. With JSTF, text-processing clients can turn glyph substitution and positioning features on and off to adjust line lengths and glyph spacing when justifying text. GDEF contains assorted tables that define useful information for text processing clients, such as ligature caret positions and lists of glyph attachment points.

So, OpenType allows advanced formatting data to be placed in additional tables. The table of interest for the character-glyph model is the ‘GSUB’ (glyph substitution) table. The text-processing client uses the ‘GSUB’ data to manage glyph substitution actions. ‘GSUB’ identifies the glyphs that are input to and output from each glyph substitution action, specifies how and where the client uses glyph substitutes, and regulates the order of glyph substitution operations. Any number of substitutions can be defined for each script or language system represented in a font. This table is analogous to the ‘mort’ table in AAT.

OpenType fonts can distinguish between script and language-specific character-glyph transformations. AAT does not currently provide direct support for this. But AAT turns out to be more ‘powerful’ than OpenType.

Firstly, AAT does not require the client to do any specific processing for certain character/glyph transformations to take place. OpenType does. That is, if a type designer comes up with a new or custom transformation specific to his or her own font, an AAT client would automatically provide support for it; an OpenType client (this can be both uniscribe and the application such as word processor) would have to be specifically revised to do so [2].

Secondly in OpenType client applications are expected to do all glyph substitution operations irrespective of any font, for a particular script and language themselves. One of the stated principles of OpenType is that writing system behavior should be handled in the application rather than in the font [3]. As much as possible, the tables of the OpenType layout define only the information that is specific to the font layout. The tables do not try to encode information that remains constant within the conventions of a particular language or within the typography of a particular script. Such information that would be replicated across all fonts in a given language belongs in the text-processing application for that language, not in the fonts [2].

Because of script specific information lying with the application and furthermore because the OpenType client application has to specifically interact with the font in order to do font-specific glyph substitutions, OpenType is not "fully intelligent".

The distinction is more this: A programmer utilizing an AAT-based interface will not need to do any of the Unicode rendering support on their own; the system will do it all. They don't need to do any of the reordering for the bi-directional algorithm, nor will they need to be aware of what language or script the text is being used to represent. But OpenType's fundamental lack is that it only partially handles the knowledge needed to perform smart rendering. A programmer utilizing OpenType, however, will have to include some of that in their program. They will have to do some of the bidirectional reordering themselves, and they will have to be aware of the language of the text so that they can select the appropriate glyph substitutions from the 'GSUB' table for processing. The OpenType model is given in figure 1 below.

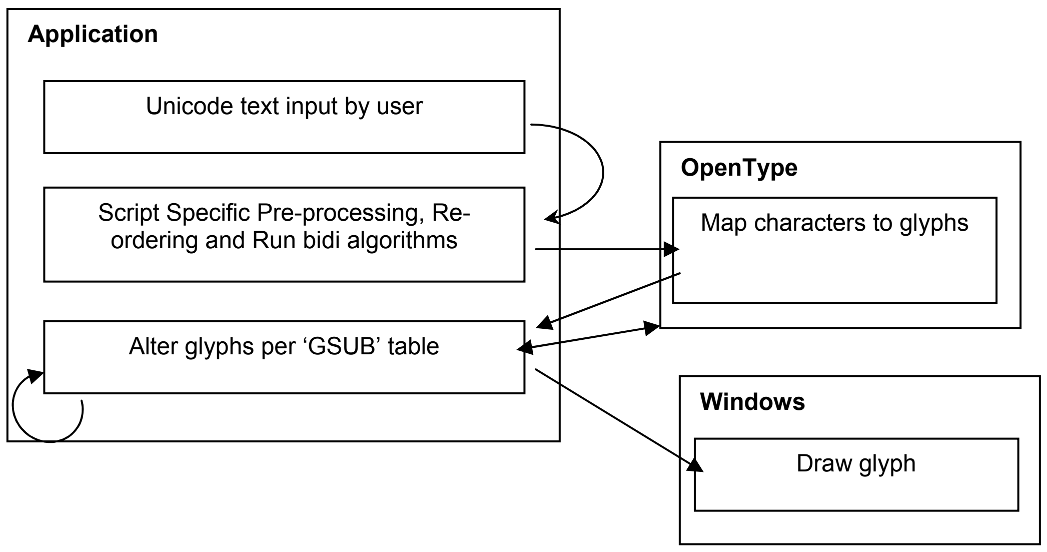

Application
Unicode text input by user
Script Specific Pre-processing, Re-ordering and Run bidi algorithms
Alter glyphs per ‘GSUB’ table
OpenType
Map characters to glyphs
Windows
Draw glyph

Figure 1 OpenType Model

An OpenType programmer has to do more work; an ATSUI programmer has less control. Both approaches are legitimate. OpenType is more designed for use by programmers who are writing their own text-drawing engines, such as is true for Microsoft Word. ATSUI is aimed at programmers who want to provide correct international support and advanced typography without writing their own text-drawing engines [11].

Microsoft’s solution to this problem is Uniscribe, a layout engine providing an API for smart script layout and rendering.

2.4. Uniscribe

Uniscribe, also commonly called Unicode Script Processor (USP), is built upon OpenType and resides in between application and OpenType. In addition to being fully capable of processing OpenType rules, it also provides support for reordering and bi-directional algorithmic implementation. There is a difference between OpenType rule extractor and OpenType rule interpreter.

The former task is actually done by OpenType Layout Services provided by Microsoft. These were specifically designed to extract information form OT tables and store it in local data structures. How and which of these rules are to be processed or 'fired' and in which order they are to be applied is done by uniscribe; This depends on the Unicode input from the application and the rules for character processing for a script are defined in the Unicode Standard; hence the name Unicode Script Processor. Thus it removes some of the weaknesses inherent in OpenType. It takes up the responsibilities of providing text layout from the application as shown in figure 2 below.

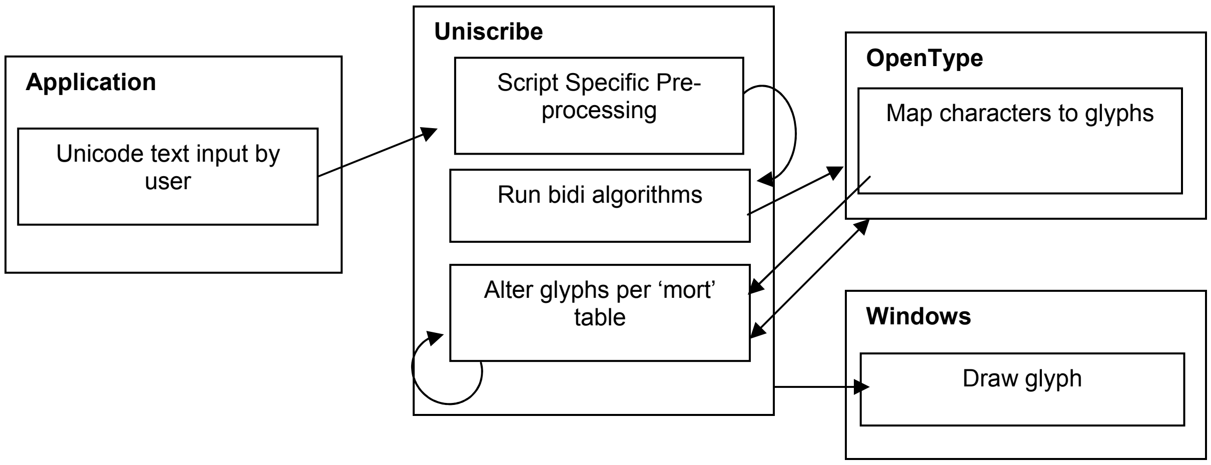


Figure 2 Text processing using Uniscribe

Uniscribe completely insulates client applications from the shaping knowledge required for complex scripting. Once uniscribe has finished calling the OTLS, it can pass the glyph string back to the application. The application meanwhile only needs to manage the original backing string of Unicode text in the logical order i.e. the order in which it was typed. Unicode never changes this backing string. Any character reordering required by Unicode script shaping rules occur in a separate buffer [6].

Unicode also implements the Unicode bidirectional (bidi) algorithm for mixed text directions. This is essential for correctly re-ordering Unicode text strings containing characters with different directional properties e.g. Arabic words with English text. It is not limited to mixing scripts however; Arabic and Hebrew are both written (read) from right to left but their digits are still written left to right.

However, Uniscribe has not been built in an extensible fashion, so no behaviors can be added or changed. While Uniscribe is slated to support rendering all of Unicode, it is not expected to provide any support for the Private Use Area. This and some other implications of the Microsoft technologies are listed next [13]:

1. Uniscribe is the proprietary of the Microsoft. It cannot be modified, upgraded or distributed depending on the scripting needs of a particular region of the world. For scripts that are supported, distribution of the script knowledge of such writing systems among the application and script processor is not uniform which should actually be. As a result uniscribe cannot be used wholly as a script processor in another platform.

2. The cost involved in providing script support has forced the software suppliers to choose, based on some marketing model of benefit vs. costs, what scripts to support. If a part of the world cannot provide sufficient purchasing power or if the script is too complex then such scripts are usually ignored and not implemented.

3. And lastly, the Private Use Area, which must be used for yet-to-be-standardized scripts or scripts related to standardized ones except for a character or two, are not supported, because doing so is too costly for the small benefit.

Macintosh provides truly extensible smart rendering capabilities. ATSUI and AAT can meet the text processing needs adequately if they were available on the Windows platform. Unfortunately, they are not, and neither is it practical for many around the world to use Macintoshes. Secondly, it is not based on the OpenType specification and is therefore beyond the scope of this proposal. The solution provided by Microsoft is, however OpenType compliant. Uniscribe removes the weakness in OpenType and hence complex script processing has reached the level of stability.

Furthermore, this provides a good typographical model, a model that can also be tried on other platforms such a Linux. The next section examines the Linux environment and explores the possibility of enabling (or enhancing) the OpenType support for Urdu.

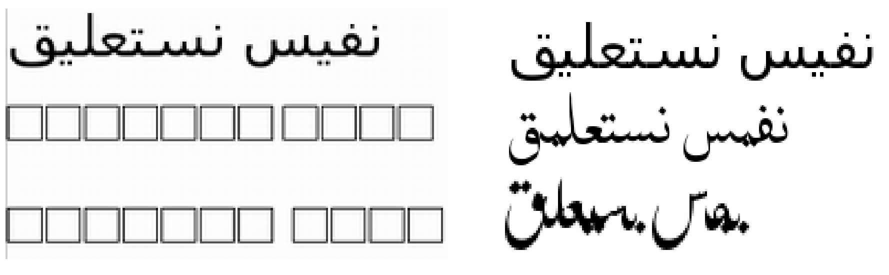
Chapter 3

OpenType support on the Linux Platform

Multi-lingual text processing is currently at its infancy in the Linux environment. Actually, in Linux nothing is structure as in Windows or Apple. It is also not very strong in terms of integration. As a result several font engines are available that may or may not be used by the applications. Since there is no ‘default’ rendering engine, developers use what they prefer. In addition, two graphical environments exist side by side in Linux, namely GNOME and KDE. These are independent of one another. GNOME has its own applications different from KDE. More important they have different rendering and text processing engines and libraries.

3.1. Rendering Engines in GNOME and KDE

As already mentioned, GNOME and KDE have separate rendering and text processing engines. GNOME relies on Pango for text layout and rendering where as KDE utilizes the QT rendering engine. The specification of each of these is quite different although both of these aim to provide OpenType support in Linux in their respective environments. A small text written in three OpenType fonts showed that GNOME has a better OpenType support than KDE as shown in figure 3 below. The font used (from top to down) were Tahoma, Nafees Naskh and Nafees Nastaleeq



(a) (b)

Figure 3 Sample text output in KDE (a) and GNOME (b)

KDE utilizes the presentation forms in Tahoma to display it correctly. Nafees fonts on the other hand are pure OpenType fonts, have no Arabic presentation forms and yield no output in KDE, indicating missing OpenType support. Things are slightly better in GNOME however. Each of the letters has acquired correct positional form (i.e. initial, medial or final form). For example letter *fay*, *seen*, *tay*, *ain*, *laam* and *choti yay* occur in medial position and acquire the medial shape. These letters however do not acquire the correct medial shapes in the given context indicating missing contextual substitution support in GNOME.

In light of the above explanation and keeping in consideration past exposure, GNOME environment was chosen over KDE for further analysis. Moving into GNOME, the two major players were FreeType and Pango. Pango was prevalent in OpenType support and text processing whereas FreeType the most popular font service engine available.

3.2. FreeType

FreeType is a rendering engine and a font service provider. FreeType can be used, modified, upgraded and distributed under the terms of the FreeType project. The FreeType team headed by Werner Lemberg and David Turner developed this library [4]. This library was designed to provide a common interface to various file formats along with improved hinting and rendering functionality.

FreeType's font file reader is capable of physically reading the contents of different font files. It can also read OpenType tables and store them in a local data structure. This information is passed on to the renderer that is involved in generating images of corresponding input characters. This is done by first locating the glyph index of input characters. Then the outline of glyph is generated. The application using FreeType simply loads/ copies the outline. This is displayed on location based on the current cursor position of the application.

Although FreeType is OpenType table 'literate', this degree of OpenType support is not enough. The engine already reads OpenType files perfectly and holds all OpenType rules within itself but what it does not do is handle or process these rules. A script and rule processing service such as uniscribe in Windows is essential for correct complex text layout. FreeType does not support such layout functionality. And as these items were precisely omitted from the library ‘by design’; there is no plan to provide such support.

As an alternate to this ‘missing layer’ in FreeType, a ‘Script Processing Service’ by the name of Pango exists side by side. It is *only* a text layout engine and uses FreeType as a font engine. This is discussed in more depth in the next section.

3.3. Pango

Pango is a layer one level higher than FreeType. Pango is a library for laying out and displaying internationalized text. It handles almost every writing system in the world, and can work on top of multiple different display systems – including traditional X fonts, or client-side OpenType fonts. Pango is used for all text handling in the soon to be released version 2.0 of the commonly used GTK+ widget toolkit [11].

Rendering of Urdu or Arabic text in not simple. Unlike English, it is not a matter of picking up the correct symbol from the font and displaying them in the order in which they occur. Had this been the case then all that is required is a font listing all the characters in the worlds according to their Unicode. But due to the complexity of different scripts this is not the case.

The first issue encountered in multi-lingual text processing is the direction of writing of the language. 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. Things get further complicated when the text in these languages can be a mixture of both right to left text and left-to-right text e.g. numbers, which according to the Unicode Standard are always written from left to right. A complicated reordering process is needed that can map the actual in-memory input string to the glyph being displayed.

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. A detailed description of Pango architecture is presented in the next chapter.

Chapter 4

The Architecture of Pango

The goal of the Pango project is to provide 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 and presents the application programmer with generic model of lines and paragraphs [11].

4.1. High Level Architecture

It consists of two major components as shown in figure 4 below. The first of these is the PangoLayout module. This module takes in a Unicode input along with a list of formatting attributes to apply to the different sections of the text and returns a glyph string after applying all the necessary script and OTF operation. The attributes usually are the font name, size, left/right aligned etc. Various properties, such as the line width, line spacing, and indentation, can also be set on the entire layout [11]. The second module referred to as the rendering component of Pango takes glyph images and gets them presented on screen.

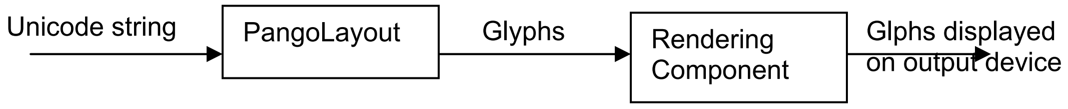


Figure 4 High Level Architecture of Pango

The PangoLayout module is responsible for the actual processing of scripts and is the centre of focus here. The PangoLayout component can discretely be further broken down into two elements. The first of these is the ‘Pango Core’ module. This contains core Pango functionalities such as itemization, line breaking etc. In addition to this it is also the central processor that manages the working and functioning of the other two modules viz. the Unicode Script processing module and a rendering system as shown in Figure 5 below.

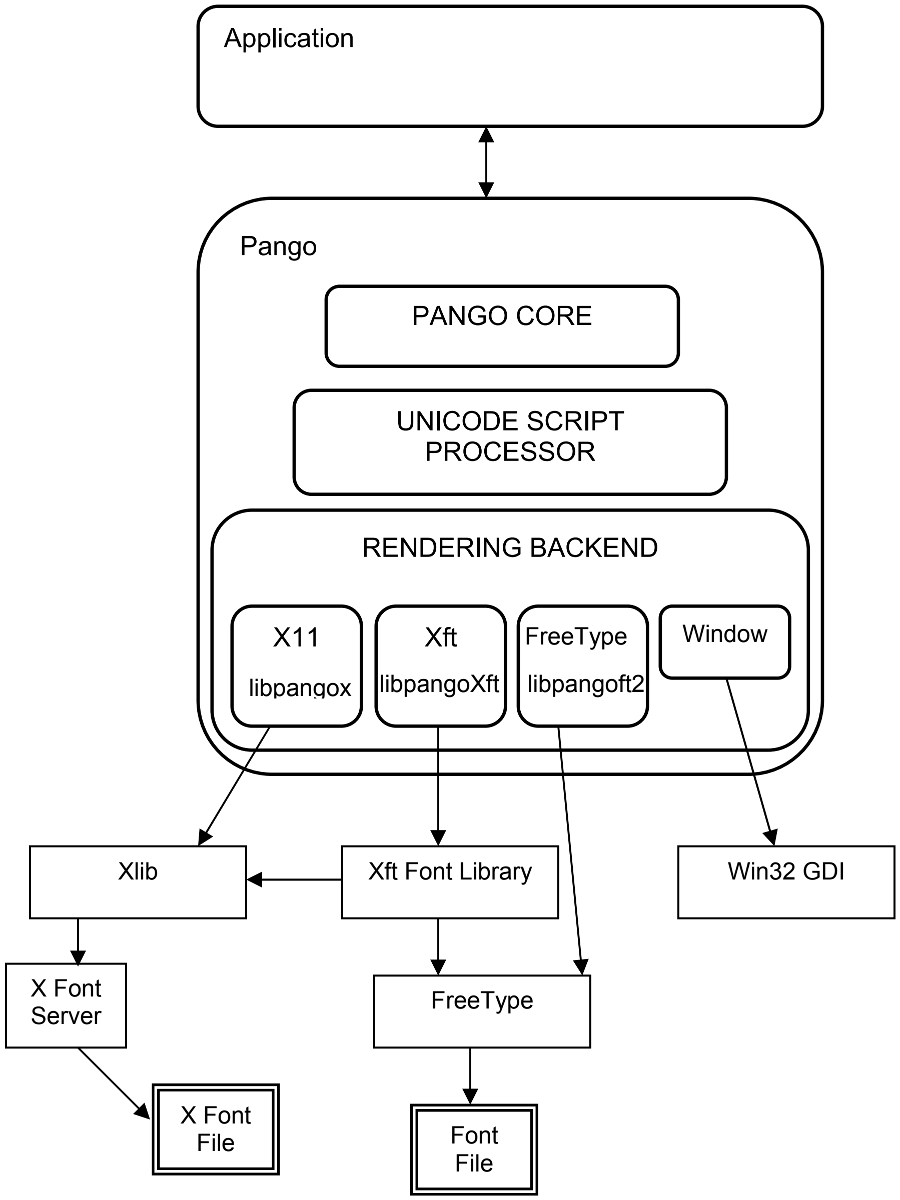


Figure 5 Detailed Architecture of Pango

4.2. Pango Core Component

The Pango Core module's basic task is to subdivide strings of characters into items (a character string having all the same script and direction attributes). Then based on the contents of each item, the text is processed. It also supports line breaking at word boundaries, hit testing and cursor positioning. Character-to-glyph mapping is provided by FreeType routines. The core module also manages bidirectional character reordering using the Unicode bidirectional algorithm.

4.3. Unicode Script Processor

After the Pango core module has divided the strings of characters into items, these items are passed over to the Unicode Script Processor. All the characters in one single item belong to the same script, according to the definition of itemization. So, depending on the script to which these characters belong to, the Unicode Script Processor applies the corresponding script processing operations on each item.

In Arabic this would be to assign a type 'base glyph' to for example Unicode 0628 (Unicode for letter *bay*) and 'mark glyph' to say Unicode input of 064E for diacritical mark *zabr*. This can further be extended to determine the positional shape of each letter that is should acquire by assigning initial, medial final or isolated tag to each letter. Similarly for Dzongkha this would be to do character reordering whereby characters must be rearranged from logical (keystroke) order to visual order. It is necessary here to make a distinction that these operations are specific to the script and are applicable to all languages belong to the same script.

The main reason for including the text layout and script semantic algorithms within Pango is to avoid unnecessary overhead in font files or applications. This relieves the font developer from having to define generalized script rules within a font.

Moving into the physical internals of the Unicode Script Processor, it comprises of separate modules for each script though these modules, also called language modules.

Each language module exists as a separate entity. Each language module can be considered as a library that is dynamically loaded when a script corresponding to that language has to be processed. As an example the Arabic module contains code specific to Arabic languages such as Arabic, Urdu etc. and is dynamically loaded by the ‘Pango Core’ when Arabic text is to be processed by the main application (such as Gedit).

4.4. Rendering Component

The ’Pango Core’ module does not handle rendering, nor does the Unicode Script Processor. The rendering component of Pango was therefore designed to do this job. The Pango rendering component is designed to support four different systems: the X11, Xft, FreeType and Windows subsystems. Each of these subsystems further provide support for traditional X fonts, for client-side fonts (both X fonts and TrueType) using the Xft library and Xrender extension [Packard], for fonts rendered locally using the FreeType library, and for fonts in the Win32 API respectively. Also, Pango uses the Xft font library and Xlib library to interface with the Xft and X11 subsystems respectively. The interface to Xlib and Xft is very inconvenient. So Pango provides its own helper libraries to interface with Xlib and Xft font library. These rendering-specific libraries i.e. libpangox*, libpangoXft*, libpangoft* and libpangowin*, included with Pango (contained in rendering module) can be compiled to support the four above mentioned subsystems.

Only recently efforts were made to integrate the four libraries i.e. libpangox*, libpangoXft*, libpangoft* and libpangowin* in to one single library. This effort was successful and the resulting Pango architecture is given in figure 6 below:

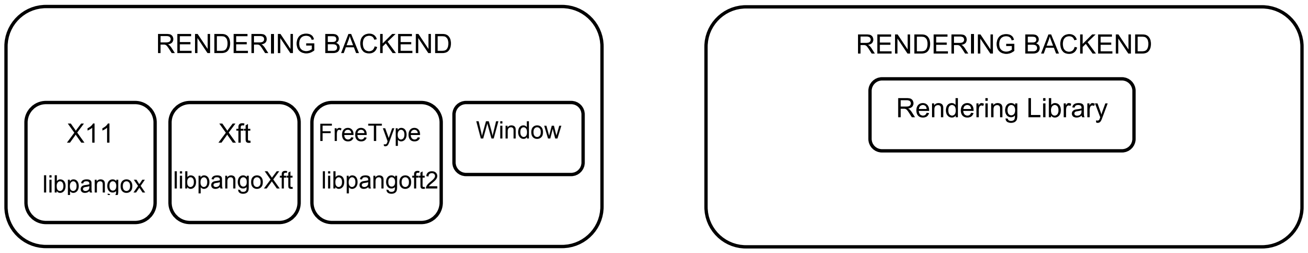


Figure 6 Integration of 4- rendering subsystems

At the present the Xft backend has the largest set, including modules for Arabic, Hebrew, Thai, Korean and 7 different Indic languages. The Xft library uses FreeType as a font engine and glib for display. It also interfaces with the Xlib, but does not require it. The Pango rendering engine has already dropped the traditional X font support in version 1.3 (development version) [11]. Moreover the Gnome environment also uses the Xft backend for rendering all TrueType fonts. As X fonts are not within the scope of this thesis, any future reference to fonts will imply TrueType fonts in general

Chapter 5

Arabic Text Processing in Pango

After discussing the architecture of Pango in depth, the discussion is extended to how Arabic text is processed in Pango and what role the three components have. This is shown in figure 7 below.

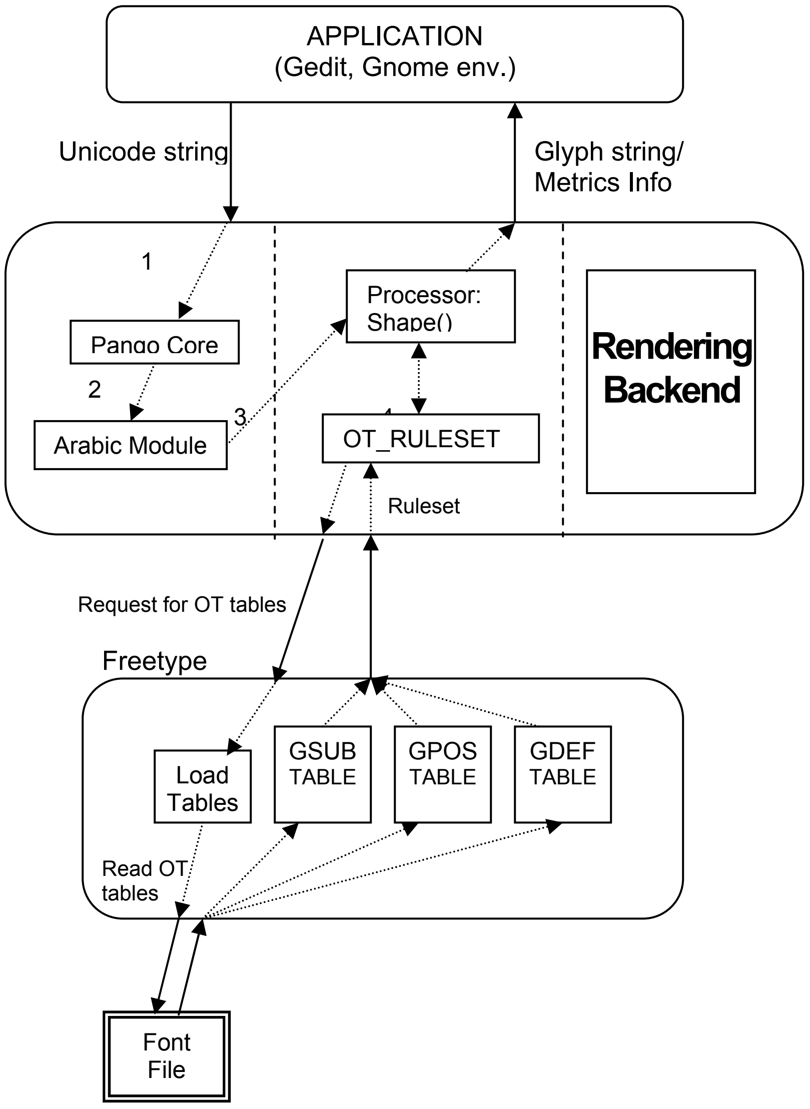


Figure 7 Arabic Text Processing in Pango

5.1. Pango Core: Itemization & Textual Boundaries Resolution

After the Unicode character string is passed over to Pango by some application say Gedit, it is sent to the 'Pango Core' module (step 1). Here the input stream is itemized. That is a character string having all the same script and direction attributes is separated. In case of rich text format, these substrings are further broken down according to their format.

After itemization, comes the boundary resolution process. Textual boundaries such as word boundaries and line breaks are determined for each item. Then the substring or item that belongs to one script, has the same format and does not have any line/paragraph break is passed over to the Unicode script processor (step 2). It first analyzes the item to determine its script, which say in this case is Arabic. Based on this analysis, the string is then forwarded to the Arabic language module contained within the Unicode Script Processor (USP).

5.2. Pango Unicode Script Module: Unicode to Positional Form

The USP is the core Script Processing Module and contains the useful script specific information. The high level architecture of USP is shown in figure 8 below. This string is sent to the shape module which transforms glyph indexes to glyphs along with its metric information. OpenType rules are also applied at this stage.

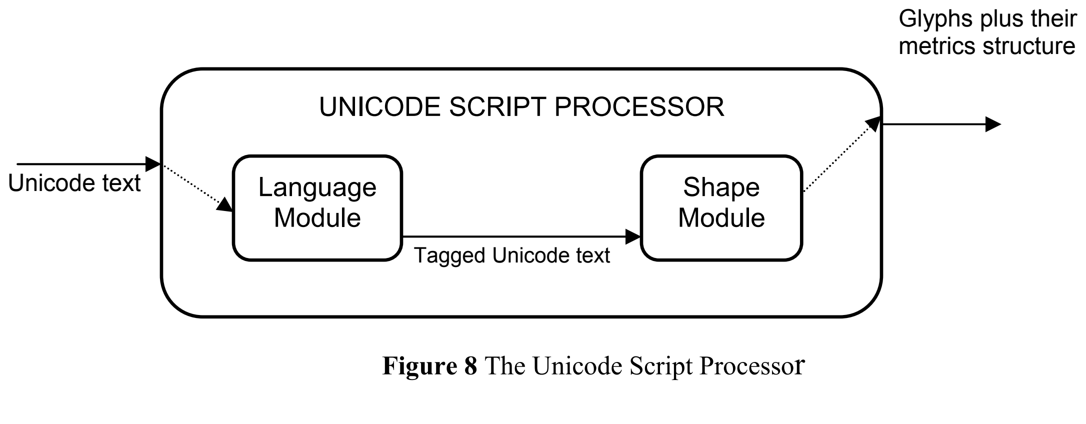


Figure 8 The Unicode Script Processor

5.2.1 Language Module

The language module performs some script dependent initial preprocessing on input string. It also coverts the Unicode input to glyph output. In Arabic this would be to assign a type 'base glyph' to for example Unicode 0628 (Unicode for letter *bay*) and 'mark glyph' to say Unicode input of 064E for diacritical mark *zabr*. It will not be wrong to say that this would actually require a large *database* containing this information. This initial processing can further be extended to determine the position of letter that is whether it is in initial, medial final or isolated place.

For e.g. to the input string *kaf*, *tay*, *alif* and *bay* (kitab), sent by the application, initial medial final and isolated tag would be assigned to *kaf*, *tay*, *alif* and *bay* respectively. In short, various properties are assigned to each character in string. A brief description of how this is done is given next.

Arabic has two different types of letters; those that can connect with both the previous letter and next letters and those that can *only* connect with the previous letter. The latter type is some time referred to as separators. In Pango terminology the former are referred to as dual while latter is named as right, because they connect from right only, the previous letter, considering Urdu is written is from right to left. Here is a sample code of how the medial property is assigned to *tay* in word *kitab* (example given in previous paragraph)

```
if ( previous == dual   )                        /* if previous is a non-seperator */
        if (current == dual)                     /* if current is a non-seperator   */
                if ( next == right   || next == dual   )  /* if next is a letter of any type  */
                    properties[i] = medial_p;    /* The current is medial            */
```

Similarly property field of letters in initial, final and isolated position is assigned the values initial_p, final_p and isolated_p respectively. The table 2 below shows the assignment.

The letter previous to *tay* should not be a separator, so the first condition simply checks the dual feature of previous letter. Then for *tay* to be classified as medial and not final, a second check is made to see that it is followed by *another letter*. This letter can be either a separator or a non separator. If however *tay* is followed by a space or vowel mark then *tay* is classified as final

Property	**Binary value**
Isolated_p	0000 1110
medial_p	0000 1101
Final_p	0000 1011
initial_p	0000 0111

Table 2 Binary Tags assigned to isolated, initial, medial and final forms

The need for such assignment becomes apparent here:

> *"According to OTF specifications, all rules are associated with one of the registered feature. These features are defined to represent a particular typographic behavior. As an example the 'init', 'medi' and 'fina' features symbolizes respectively the initial, medial or final character behavior in Arabic. Therefore all initial, medial and final rules are specifically listed under the respective features."*

Now a letter can occur at and have a rule defined for each initial, medial, final and isolated position. If the text processor encounters this letter in the input string then which of these rules should be applied. This is where this 'pre-processing' helps. Referring back to example of *kaf*, assigning initial tag to '*kaf*' helps the text layout engine to determine that the possible substitution rules to be applied on '*kaf*' are to be looked for under the 'init' feature. Consider it the other way around, assigning initial tag to '*kaf*', informs the text layout engine that it should not apply the rules under the 'medi', 'fina' and 'isol' feature. In short, it saves language and script information within the text stream to clearly associate character codes with typographical behavior. Once the properties are assigned, the text processing begins (step 3 in figure 7).

5.2.2 Shape Module

The shape module can be considered the central processing unit for Pango. It the designed to process the Unicode text after it has been properly pre-processed by the language module. This involves 'searching' the OpenType tables (that have already been loaded in to the OT_RULESET structure of Pango through the FreeType library) to get the rules for the corresponding input string. If a match is found the corresponding rule is processed. Throughout this process Pango keeps track of the association between the character codes for the original text and the glyph indices of the final text.

TrueType Level Processing

Applying the corresponding OTF rules to the tagged Unicode input string to convert it to output glyphs is done by the shape module of the USP. This conversion is done in three steps. The first step is exactly like how English text is processed in TrueType environment. That is each Unicode character is mapped onto its corresponding glyph index. These glyph indexes usually (or always) contain the isolated forms of the input characters. The mapping of character codes to the glyph index values used in the font is defined in the cmap table. It may contain more than one sub table, in order to support more than one character-encoding scheme. Character codes that do not correspond to any glyph in the font are mapped to glyph index 0. The glyph at this location must be a special glyph representing a missing character.

OpenType Level Processing

Once all the Unicode characters have been translated to 'default' glyph indexes, the new 'glyph string' along with its properties string (tagged information) is now ready for OpenType processing viz. the second step of shape module. The glyph string and the properties string are collectively called PangoString in Pango. The processing is done as follows: Each lookup is applied to each letter in the string. Lookup is a very popular sub table in the OTF formalism. A brief description is given next.

The information used to substitute and position glyphs is defined in Lookup sub tables. Each sub table supplies one type of information, depending upon whether the lookup is part of a GSUB or GPOS table. For instance, a GSUB lookup might specify the glyphs to be substituted and the context in which a substitution occurs, and a GPOS lookup might specify glyph position adjustments for kerning. More can be read from the OTF specification at http://www.adobe.com/type/opentype/main.html.
All initial rules pertaining to all the characters in a given script are listed under one lookup tables. Similarly all medial and final rules are specified in separate lookup tables. Each of these lookups is associated with the 'init', 'medi' and 'fina' features respectively. Also, all these lookup tables combine to form the TTO_lookuplist. The structure of TTO_lookuplist, according to OT specification is:

```
struct  TTO_LookupList
  {
    FT_UShort    LookupCount;          /* number of Lookups       */
    TTO_Lookup*  Lookup;               /* array of Lookup records */
  };
```

At present, with the exception of the four ('isol', 'init', 'medi' and 'fina') features already discussed, Pango does not support any other OT features. It therefore does not recognize those lookups that are associated with the other features such a 'liga' and 'calt'. So, coming back to the working of the shape module, each of the lookup, recognized by Pango, is tried on every element of the 'glyph string'. At this point the properties string, that has the tags for each input character is utilized to determine for which character which lookup should be allowed.

As an example if a particular character 'C' has an initial_p tag, then the lookup 'I' associated with the 'init' feature is *allowable* for that character. That is, the rules for 'C' are to be searched in ‘I’. If such a rule is found it is ‘fired’ and the shape of letter is transformed from default to its initial form. The lookups associated with the ‘medi’ or ‘fina’ feature become unallowable for ‘C’ and are simply ignored. How is this comparison made?

A third field is added to the TTO_lookuplist. This is the properties field.

```
 struct  TTO_LookupList
 {
   FT_UShort    LookupCount;        /* number of Lookups           */
   TTO_Lookup*  Lookup;             /* array of Lookup records     */
   FT_UShort*   Properties;         /* array of flags              */
 };
```

The `Properties' field is not defined in the TTO specification but is needed for processing lookups. Every lookup has a property field. The table 3 below shows the assignment.

Lookup	**Properties Binary value**
Isolated	0000 0001
Medial	0000 0010
Final	0000 0100
Initial	0000 1000

Table 3 Binary Tags assigned to lookups for isolated, initial, medial and final forms

Now all that is required is to process Lookup [n] for glyphs that have the specific bit not set in the `properties' field of the input string object. Finally in the third step, the shape module translates the glyph indexes to glyph outlines. Then these outlines are transformed to bitmaps. The glyph shapes along with the position/metric information are passed over to the rendering module. The rendering module displays them on screen.

PART TWO

The Problem

Chapter 6

The Alternating Thick-Thin-Thick Joins

Initially Pango fully supported the four ‘isol’, ‘init’, ‘medi’ and ‘fina’ features already discussed. Its also provide support for contextual substitution through the OpenType Glyph Substitution Table type 7 viz. Chain Contextual Substitution. In addition to this OpenType also specifies another table. This is Reverse Chain Contextual Single Substitution table type 8. This table was not yet incorporated into Pango. It was however useful in modeling the left-to-right context dependency drifts in complex script like Nastaleeq. Implementation of this table will enable the modeling of Urdu scripts according to its natural way of writing.

It describes single glyph substitutions in context with an ability to look back and/or look ahead in the sequence of glyphs. The major difference between this and other lookup types is that processing of input glyph sequence goes from end to start. This format is restricted to only coverage based subtable format. Also only single substitution allowed on input glyph that matched one of the glyphs in the Coverage table and has the context according to BacktrackCoverage and LookaheadCoverage (if applicable).

6.1. Usefulness

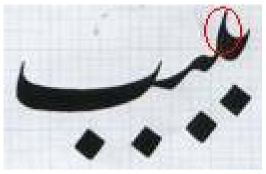
Its usefulness can best be understood by analyzing some of the properties prevalent in the common Urdu writing systems.

6.1.1 Variation in Shape of Join to Break Monotony

It is possible that many similar joins come together in a ligature, as in many Bay’s connected to each other. In such a case, the similar shape of the joins 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, 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.

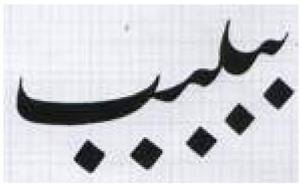
The principle is illustrated below with some examples.



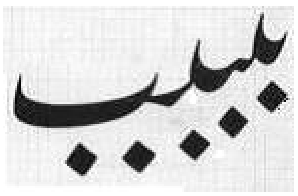
The raised join is shown in red circle.

6.1.2 Frequency of Usage of Raised Joins

It has been mentioned in the above section that 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 unraised joins is almost equal. Hence, a ligature can in principle be formed in either of the two ways: raised joins coming in between unraised ones and unraised joins coming in between raised ones. However, in practice, the first one is preferred as discussed below.



Starting with the unraised join, every second one is raised.

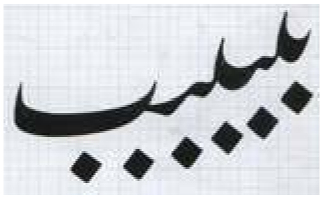


Starting with the raised join, every second one is unraised.

6.1.3 Minimization of Number of Raised Joins

The alternation of raised and unraised cusp-like joins is to be preferably constrained in such a way that the number of raised cusps is minimized.

This principle is illustrated with the following examples.

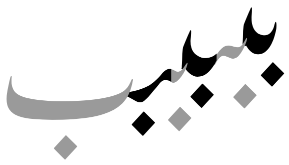


When six Bay's come together, the number of raised joins is minimized if we start with a raised join.
Note: Same is true for Naskh script.

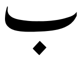
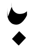
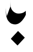
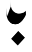
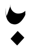
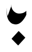
6.2. Modeling Using OTF Technology

Although such variation can be modeled using the GSUB Lookup type 6 viz. chaining contextual substitution of the OpenType specification, this has certain limitations.

Consider the following word:



This has six different shapes. Starting from left to right these can be named as FinalBay, bayMediBfBayFinal, bayMediTiny, bayMediThick, bayMediThin, bayInitThick. Let's try to define contextual alternate (calt) rules for this word using contextual chain substitution table. Assume initially all medial and initial bays have shape bayMedi1 and bayInit1 respectively, acquired after the application of the 'medi' and 'init' feature.

					
FinalBay	bayMedi1	bayMedi1	bayMedi1	bayMedi1	bayInit1

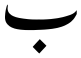
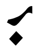
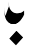
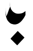
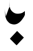
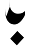
Lookup 1

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

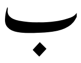
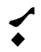
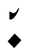
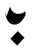
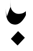
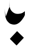
Lookup 2

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

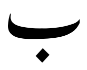
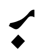
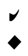
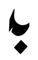
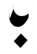
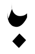
Lookup 3

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.

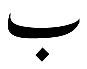
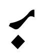
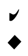
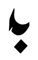
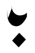
bayMedi1 → bayMediThin

Similarly for initial bay we can have

bayInit1 → bayInitThin

whenever it is followed by bayMediThick

The string now looks like:

					
FinalBay	bayMediBfBayFinal	bayMediTiny	bayMediThick	bayMediThin	bayInit1

Lookup 5

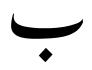
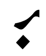
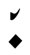
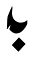
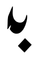
And finally for the initial bay we have

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 lookup1	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 lookup1	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.

Chapter 7

The Lookup Processing Algorithm at Work

7.1. The 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)

That is all lookups are applied on all the glyph. This makes the algorithm very inefficient. Consider a situation when a glyph has absolutely no substitution rule but it is still tested and tried on all lookups. Things get worse when a font like Nafees Nastaleeq has a huge number of substitution lookups. A glyph is rigorously looked through over some 100 lookups and if this glyph has no substitution rule defined then one can imagine the text processing engine wasting lot of time doing *nothing.*

7.2. Experiment: Determining the Hit Ratio

A small experiment was conducted to find out the hit ratio. Of the total number of times a lookup is searched for a glyph, Hit is defined to be a situation where:

1. Glyph is searched in a lookup AND
2. A substitution rule for that glyph is listed in the lookup AND
3. The look-ahead and back-track context in lookup matches the context of the glyph

i.e. a glyph is successfully substituted when the given lookup is applied on it.

7.2.1 Nature of Data

Data that was used for the experiment was all valid 3-character and 4-character ligatures listed in [12]. There are about 3000 3-characters ligatures and more than 6000 4-character ligatures. The fonts used were Nafees Nastaliq and Nafees Naskh.

7.2.2 Methodology and Result

For each font and for every ligature the hit (or success) and trail were calculated. This can be explained with a small example. Consider a ligature *bay-bay-alif*. After application of the initial, medial and final lookup the resulting input string is:

Bayinit1	onenuqtabelow	baymedi1	onenuqtabelow	aliffina

Now each glyph in the string is tried on 104 lookups (in Nafees Nastaliq). The total number of accesses are 102 * 5 = 510. In this ligature of all the glyphs, only bayinit1 is substituted for another glyph. Thus of the 510 time a glyph was searched in a lookup, the number of times a substitution took place or there was a success was only once.

For the entire data set it was observed that that the hit ratio is well below 0.50%. (For the above data the hit percentage is 0.19% only.) This detailed analysis shows that this algorithm has lot of overhead and can be improved.

Chapter 8

The Problem Statement

"Enhancing Contextual Substitution Support for Urdu in Pango Using OpenType"

As have already been discussed, the OpenType Support in Linux is in a pre-mature stage. This limited support is available through the Pango text processing library in the GNOME environment. Also, *some* contextual substitution is possible but this is still not adequate enough to properly layout Urdu (scripts). For example, it could not process the context sensitive substitution rules essential for correct rendering of Nafees Nastaliq and Nafees Naskh scripts

Furthermore, writing styles commonly used for Urdu language are highly context sensitive and consequently their full realization requires a huge contextual substitution grammar. The lookup processing algorithm, specified by OpenType specifications, is very inefficient for large grammar set. This makes the processing of OpenType fonts that model and implements this grammar, very slow and inconvenient for use.

This enhancement of contextual substitution support in Pango as outlined by the problem statement will therefore focus on two dimensions. The first dimension would be the physical expansion towards completeness of substitution support in Pango library. In the first phase, the OpenType tables that had already been implemented by the Pango Research Group, but were inconsistent and buggy will be fixed and patched. In addition a new OpenType Glyph Substitution type 8 Reverse Chain table which was specifically designed to support the left-to-right context dependency drift in scripts like Nastaliq, will be added to total the OpenType substitution support in Linux.

The other dimension is performance that is the efficiency in processing the substitution tables. In this phase the OpenType rule processing algorithm, proved to be very inefficient, will be improved to provide a more efficient mechanism for processing the lookups on the input string.

PART THREE

The Proposed Solution

Chapter 9

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. This has already been explained in chapter 5. The solution to the missing Reverse chain table is to implement one.

Reverse Chaining Contextual Single Substitution subtable describes single glyph substitutions in context with an ability to look back and/or look ahead in the sequence of glyphs. The major difference between this and other lookup types is that processing of input glyph sequence goes from end to start. This format is restricted to only coverage based subtable format. Also only single substitution allowed on input glyph that matched one of the glyphs in the Coverage table and has the context according to BacktrackCoverage and LookaheadCoverage (if applicable).

Before moving on to the design and implementation of this table let us re-look the problem in chapter 6 and how introducing a reverse chain lookup solves the problem.

9.1. Introducing a Reverse Chain Lookup

Consider the 5 lookups specified in chapter 5. Clearly these were unable to model the thin-thick-thin variation in Nastaleeq. Using reverse chain table however makes it possible remove this discrepancy. 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 from contextual chain table to reverse chain table.
2. bayMedi1 → bayMediThick

   whenever it is followed by bayMediTiny or

   whenever it is followed by baymedi1 baymediThick

9.2. 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.

9.2.1 Adding Reverse chain substitution table to GSUB table

The current glyph substitution capacity of Pango extends till substitution type 6. Type 7 is the for extension table. The type 8 reverse chain table was included in GSUB table by adding

```
        TTO_ReverseChainContextSubst        reverse;
```

The reverse chain contextual substitution table as specified by OT is given in table 4 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 4 ReverseChainSingleSubstFormat1 subtable: Coverage-based Reverse Chaining Contextual Single Glyph substitution .

The corresponding structure for the GSUB Lookup type 8 i.e. Reverse Chaining Contextual Substitution format 1 table is given below. The data fields are according to the Freetype 2 conventions. FT_UShort is redefinition of unsigned integer (2 bytes).

TTO_Coverage is also a data structure for holding the coverage information contained in the coverage table.

The subtable given above contains Coverage table for input glyph and Coverage table arrays for lookahead and backtrack sequences, also count of output glyph indices in the Substitute array (GlyphCount), and a list of the output glyph indices. The Substitute array must contain the same number of glyph indices as the Coverage table. To locate the corresponding output glyph index in the Substitute array, this format uses the Coverage Index returned from the Coverage table.

```
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,
}

Struct ReverseChainContextSubst
{
       ReverseChainContextSubstFormat1 rccf1;
}
```

9.2.2 Loading the Reverse chain substitution table

```
Routine 1:
  FT_Error Load_ReverseChainContextSubst(
                           TTO_ReverseChainContextSubst* rccs,
                           FT_Stream stream )
  {

    FT_Error error;

    if ( ACCESS_Frame( 2L ) )                 /* Read 2 bytes of data */
      return error;
     rccs->SubstFormat = GET_UShort();         /* First two bytes  is the
                                                        format of table */

    FORGET_Frame();

    switch ( rccs->SubstFormat )
    {
       /* Reverse chain currently has format 1 */
       case 1:
       return Load_ReverseChainContextSubst1( &rccs->rccsf.rccsf1, stream );
      default:
       return TTO_Err_Invalid_GSUB_SubTable_Format;
    }
  }
```

Routine 2:

```
  static FT_Error  Load_ReverseChainContextSubst1(
                            TTO_ReverseChainContextSubstFormat1* rccsf1,
                                                           FT_Stream stream)
  {
    FT_Error error;
    FT_Memory memory = stream->memory;

    FT_UShort                m, count;
    FT_UShort               nb = 0,  nl = 0;
    FT_UShort               backtrack_count, lookahead_count;
    FT_ULong                cur_offset, new_offset, base_offset;

    TTO_Coverage*           b;
    TTO_Coverage*           l;
    FT_UShort*              sub;

    base_offset = FILE_Pos() - 2;

    if ( ACCESS_Frame( 2L ) )
      return error;

    new_offset = GET_UShort() + base_offset;

    FORGET_Frame();

    cur_offset = FILE_Pos();

    if ( error = Load_Coverage( &rccsf1->Coverage, stream )  != TT_Err_Ok )
      return error;

    /* Input coverage loaded */
    if ( ACCESS_Frame( 2L ) )
      goto Fail5;


    rccsf1->BacktrackGlyphCount = GET_UShort();
    FORGET_Frame();

    rccsf1->BacktrackCoverage = NULL;

    backtrack_count = rccsf1->BacktrackGlyphCount;

    if ( ALLOC_ARRAY( rccsf1->BacktrackCoverage, backtrack_count,
                      TTO_Coverage ) )
      return error;

    b = rccsf1->BacktrackCoverage;

    for ( nb = 0; nb < backtrack_count; nb++ )
    {
      if ( ACCESS_Frame( 2L ) )
        goto Fail4;

      new_offset = GET_UShort() + base_offset;

      FORGET_Frame();
      cur_offset = FILE_Pos();
      if ( error = Load_Coverage( &b[nb], stream ) != TT_Err_Ok )
        goto Fail4;
    }

    /* Backtrack coverage loaded */

    if ( ACCESS_Frame( 2L ) )
      goto Fail4;


    rccsf1->LookaheadGlyphCount = GET_UShort();
    FORGET_Frame();

    rccsf1->LookaheadCoverage = NULL;

    lookahead_count = rccsf1->LookaheadGlyphCount;

    if ( ALLOC_ARRAY( rccsf1->LookaheadCoverage, lookahead_count,
                      TTO_Coverage ) )
      goto Fail4;

    l = rccsf1->LookaheadCoverage;

    for ( nl = 0; nl < lookahead_count; nl++ )
    {
      if ( ACCESS_Frame( 2L ) )
        goto Fail2;

      new_offset = GET_UShort() + base_offset;

      FORGET_Frame();

      cur_offset = FILE_Pos();
      if ( error = Load_Coverage( &l[nl], stream )!= TT_Err_Ok )
        goto Fail2;
    }

    /* Lookahead coverage loaded */

    if ( ACCESS_Frame( 2L ) )
      goto Fail2;

    rccsf1->GlyphCount = GET_UShort();
    FORGET_Frame();

    rccsf1->Substitute = NULL;

    count = rccsf1->GlyphCount;

    if ( ALLOC_ARRAY( rccsf1->Substitute, count,
                      FT_UShort ) )
      goto Fail2;

    sub = rccsf1->Substitute;


    if ( ACCESS_Frame( count * 2L ) )
      goto Fail1;

    /* Loading Substitute */
```

```
    int ncount = 0;
    for(ncount = 0; ncount < count; ncount++)
      sub[ncount] = GET_UShort();

    FORGET_Frame();

    return TT_Err_Ok;

  }
```

It is evident from the above routines that the reverse chain table is loaded dynamically depending on the backtrackcount, lookaheadcount and glyphcount. As a result a routine is required that releases this memory. Such routine is given below.

```
Routine 3:
  static void  Free_ReverseChainContext1(
                              TTO_ReverseChainContextSubstFormat1* rccsf1,
                                                          FT_Memory memory)
  {
    FT_UShort      n, count;
    TTO_Coverage*  c;

    /* Freeing LookaheadCoverage */
    if ( rccsf1->LookaheadCoverage )
    {
      count = rccsf1->LookaheadGlyphCount;
      c     = rccsf1->LookaheadCoverage;

      for ( n = 0; n < count; n++ )
        Free_Coverage( &c[n], memory );

      FREE( c );
    }

    /* Freeing BacktrackCoverage */
    if ( rccsf1->BacktrackCoverage )
    {
      count = rccsf1->BacktrackGlyphCount;
      c     = rccsf1->BacktrackCoverage;
      for ( n = 0; n < count; n++ )
        Free_Coverage( &c[n], memory );

      FREE( c );
    }

    /* Freeing InputCoverage */
    Free_Coverage( &rccsf1->Coverage, memory );

  }
```

9.2.3 Applying Reverse chain substitution table on input string

Lookups that fall under the ‘calt’ feature are applied to all the glyphs in the input string, irrespective of whether the substitution table is contextual substitution type 5, chain contextual substitution table type 6 or reverse chain contextual substitution table type 8. The generic algorithm for processing the lookups on input is quite simple.

```
Starting from lookup LK = 1 to lookupcount
    Starting from element IND = 1 to length(input string)
        Apply lookup LK on IND^th index of input string
```

Although this algorithm is applicable for all type from 1 to 6, this may not be correct from the reverse chain substitution table type 8. In this table the processing of input glyph string goes from end to start.

One solution is to have separate processing mechanism for this table. But as is discussed next this algorithm suffices for substitution type 8. 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:

new_IND = length (input string) - (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 routine that implements the manner in which lookups are to be applied is given next. This is followed by a function that actually processes the reverse chain lookup on the input string.

```
Routine 4:
  static FT_Error  Do_String_Lookup( TTO_GSUBHeader* gsub,
                                     FT_UShort lookup_index,
                                     TTO_GSUB_String* in, TTO_GSUB_String* out)
  {
    FT_Error  error, retError = TTO_Err_Not_Covered;

    /* Used a mirror for in->pos in case of LookupType 8 */
    FT_UShort new_in_pos;

    FT_UShort*  properties = gsub->LookupList.Properties;
    FT_UShort*  p_in       = in->properties;
    FT_UShort*  s_in       = in->string;

    int      nesting_level = 0;
    while ( in->pos < in->length )
    {
      if ( ~p_in[in->pos] & properties[lookup_index] )
      {
        /* 0xFFFF indicates that we don't have a context length yet */
        error = Do_Glyph_Lookup( gsub, lookup_index, in, out,
                                 0xFFFF, nesting_level );
        if ( error )
       {
         if ( error != TTO_Err_Not_Covered )
           return error;
       }
       else
         retError = error;
      }
      else
        error = TTO_Err_Not_Covered;

      if ( error == TTO_Err_Not_Covered )
      {
       /* LooupType 8 works from start to end */
       if(gsub->LookupList.Lookup[lookup_index].LookupType ==
                                                 GSUB_LOOKUP_REVERSE_CHAIN)
       {
         new_in_pos = in->length - (in->pos + 1);
         if ( ADD_String_ReverseOrder( in, 1, out, 1,&s_in[new_in_pos],     Note 1
                                                            0xFFFF, 0xFFFF) )
           return error;
       }
       else
         if ( ADD_String( in, 1, out, 1, &s_in[in->pos], 0xFFFF, 0xFFFF ) )
           return error;
      }

    }

    return retError;
  }
```

Note 1: If for a particular glyph the lookup does not apply due to mismatch of any coverage, then simply copy the current glyph into the output string. In case of reverse chaining the glyph should be placed on the appropriate location of the output string.

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. If backtrack count >0
      Match the backtrack coverage with the glyph sequence immediately preceding
      the current glyph
              If match is not made
                   return error
2. Match the input coverage with the current glyph sequence
              If match is not made
                   return error
3. If lookahead count >0
      Match the lookahead coverage with the glyph sequence immediately
      following the current glyph
              If match is not made
                   return error

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 )
  {

    FT_UShort        index,input_index, i, j, curr_pos, property, new_in_pos;
    FT_UShort        bgc, lgc;
    FT_Error         error;
    FT_UShort*       s_in;

    TTO_Coverage*    bc;
    TTO_Coverage*    lc;
    TTO_GDEFHeader*  gdef;


    gdef = gsub->gdef;

    if ( CHECK_Property( gdef, in->string[in->pos], flags, &property ) )
      return error;

    bgc = rccsf1->BacktrackGlyphCount;
    lgc = rccsf1->LookaheadGlyphCount;

    if ( context_length != 0xFFFF && context_length < 1 )
      return TTO_Err_Not_Covered;

    /* check whether context is too long; it is a first guess only */

    if ( bgc > in->pos || in->length - in->pos + lgc > in->length )
      return TTO_Err_Not_Covered;
    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 ChainingContextSubst we had
        for ( i = 0, j = in->pos - 1; i < bgc; i++, j-- )*/

      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;

    /* we are 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;
    }

    FT_Short val[1];
    val[0] =  rccsf1->Substitute[input_index];

    ADD_String_ReverseOrder(in, 1, out, 1,val,0xFFFF,0xFFFF);
    return 0;
  }
```

Chapter 10

Enhancement in Lookup Processing Algorithm

The lookup processing algorithm

"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) [1]

A detailed analysis shows that this algorithm has lot of overhead. All lookups are applied on all the glyph. Consider a situation when a glyph has absolutely no substitution rule but it is still tested and tried on all lookups. Data collected from applying all possible 3-character and 4-character ligatures on two Urdu fonts Nafees Naskh and Nafees Nastaleeq show that the percentage of successful trail of lookup is well below 0.75%.

An alternate to this mechanism was an introduction of a glyph-lookup matrix. A two-dimensional array in which the rows are equal to number of glyphs in the font while columns equal to number of lookups. A row j in this matrix listed all the lookups that had the rule for the glyph j; In order to limit the size of matrix, the glyph that did not have any contextual substitution rule had no column. But if a glyph had at least one lookup table specifying a rule for that lookup, then a column equal to number of lookups was dynamically allocated. If glyph j only had a rule in lookup n then the glyph-lookup matrix [j][n] = 1 and rest of the elements in the jth row have a 0.

Before a lookup is tried or accessed for a given input, this matrix is referred. As a result lookup is only searched if it does indeed have a rule for the current glyph. Also, although the algorithmic complexity is still number of lookups * number of glyphs, the number of accesses to lookups is reduced, eliminating the in-lookup search overhead.

10.1. Implementation

Just like reverse chain lookup type 8, the modification to the current lookup processing algorithm was done in three steps. The first step is to include a structure for the glyph-lookup matrix. The second step is to load the table and the third step is the application of this table during processing on the input glyph string.

10.1.1 Loading the glyph-lookup matrix

```
Routine 6:
void fill_Matrix_Ccf3( TTO_GSUBHeader* gsub,
                                 TTO_ChainContextSubstFormat3* ccsf3,
                                 FT_UShort lookup_index,
                                 FT_UShort** glyph_lookup_map)
  {

    TTO_RangeRecord* rr;
    FT_UShort RCount, range,i,j, igc, glyph_index;
    TTO_Coverage* ic;

    igc = ccsf3->InputGlyphCount;
    ic = ccsf3->InputCoverage;
    for(i = 0; i < igc ; i++)
      {
      switch(ccsf3->InputCoverage[i].CoverageFormat)
        {
        case 1:
          for( RCount = 0; RCount < ic[i].cf.cf1.GlyphCount; RCount++)
            {
             glyph_index =ic[i].cf.cf1.GlyphArray[RCount];
             if(!glyph_lookup_map[glyph_index])
               {
                 glyph_lookup_map[glyph_index] =
                    malloc(gsub->LookupList.LookupCount * sizeof(FT_UShort));
                 if(!glyph_lookup_map[glyph_index])
                   printf("NOT ENOUGH MEMORY\n");
                 for(j = 0; j < gsub->LookupList.LookupCount; j++)
                      glyph_lookup_map[glyph_index][j] = 0;
               }
              glyph_lookup_map[glyph_index][lookup_index] = 1;
            }
            break;

        case 2:
          rr = ic[i].cf.cf2.RangeRecord;
          for( RCount = 0; RCount < ic[i].cf.cf2.RangeCount; RCount++)
            {
             for(range = rr[RCount].Start; range <= rr[RCount].End; range++)
               {
                 if(!glyph_lookup_map[range])
                   {
                    glyph_lookup_map[range] =
                    malloc(gsub->LookupList.LookupCount * sizeof(FT_UShort));
                    for(j = 0; j < gsub->LookupList.LookupCount; j++)
                      glyph_lookup_map[range][j] = 0;
```

```
                  }
                glyph_lookup_map[range][lookup_index] = 1;
              }
            }
          break;

        }
      }
  }
```

10.1.2 Applying the glyph-lookup matrix

As have already been discussed above before a lookup is tried or accessed for a given input, this matrix is referred so that only that lookup is searched tat does indeed have a rule for the current glyph.

```
Routine 4:
while ( in->pos < in->length )
{
  if ( ~p_in[in->pos] & properties[lookup_index] )
  {
    switch(gsub->LookupList.Lookup[lookup_index].LookupType){
       case GSUB_LOOKUP_CHAIN:
       {
         if( glyph_lookup_map[in->string[in->pos]] &&
                        glyph_lookup_map[in->string[in->pos]][lookup_index])
                        error = Do_Glyph_Lookup( gsub,
                        glyph_lookup_map[in->string[in->pos]]
                        && glyph_lookup_map[in->string[in->pos]][lookup_index] )
                        * lookup_index, in, out, 0xFFFF, nesting_level );

         else
           error = TTO_Err_Not_Covered;
             break;

       }

       default:
         error = Do_Glyph_Lookup( gsub, lookup_index, in, out,
                                  0xFFFF, nesting_level );
    }

    if ( error )
    {
       if ( error != TTO_Err_Not_Covered )
         return error;
    }
    else
      retError = error;
  }
  else
    error = TTO_Err_Not_Covered;

  if ( error == TTO_Err_Not_Covered )
  {
    /* LooupType 8 works from start to end */
    if(gsub->LookupList.Lookup[lookup_index].LookupType ==
                                                     GSUB_LOOKUP_REVERSE_CHAIN)
    {
       new_in_pos = in->length - (in->pos + 1);
      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;
  }

}
```

Chapter 11

Dependent and Independent Lookups

Although the algorithm stated above yields better results than the original algorithm, it can still be improved. Looking at the original algorithm:

```
Starting from lookup LK = 1 to lookupcount
    Starting from element IND = 1 to length(input string)
        Apply lookup LK on IND^th index of input string
```

And the newly proposed algorithm

```
Starting from lookup LK = 1 to lookupcount
    Starting from element IND = 1 to length(input string)
        If(lookup LK has a rule for element at index IND)
            Apply lookup LK on IND^th index of input string
```

The number of iterations in both these cases is still lookupcount multiply by the length of input string. There should be some way, which there is, by which this can be reduced.

One such option is two divide the lookups into two categories. Independent and dependent lookups. Let’s say that there is some way by which we can divide the lookups into independent lookups that can occur independently of other lookups and no other lookups have any impact on the output of these lookups. Then there are dependent lookups. As the name suggests these are dependent on the output of other lookups. The first step would therefore be the identification of dependent and independent lookups. A small description is given next.

11.1. Determining the ‘Dependent’ and ‘Independent’ lookups

A contextual alternate lookup is usually of the configuration:

$X \rightarrow Y$

Preceded by A and/or followed by B

Where X is the outTransitionGlyph, Y is the inTransitionGlyph, A is the precedingContext and B the followingContext.

A lookup can be further divided into four types depending on the type of the outTransitionGlyph, precedingContext and followingContext.

- **Lookup configuration 1**
  The outTransitionGlyph is a default glyph[1] and the context also comprise of default shapes.
- **Lookup configuration 2**
  The outTransitionGlyph is a default glyph but the context comprise of non-default shapes.
- **Lookup configuration 3**
  The outTransitionGlyph is a non-default glyph(s) but the context comprise of default shapes.
- **Lookup configuration 4**
  The outTransitionGlyph is a non-default glyph and the context also comprise of non-default shapes.

Lookups that fall under the Lookup configuration 3 & 4 can be termed as 'dependent lookups'. Simply because these lookups have, as outTransitionGlyph, non-default glyphs and are therefore dependent on lookups that have these (similar non-default) glyphs as inTransitionGlyph. Similarly *lookup configuration 2* lookups are dependent lookups because of the same reason.

[1] Default shapes are shapes that a letter has after the 'init', 'medi' and 'fina' lookups have been applied.

Meanwhile lookups that belong to Lookup configuration 1 can be either 'dependent or independent lookups'. This is explained below with a small example.

Consider the following lookup L1:

Lookup L1:
bayinit1 → bayinit3
whenever it if followed by jeemmedi1

Here bayinit1 and jeemmedi1 are default initial and medial shapes respectively. This lookup is an independent lookup if the context glyph(s) i.e. jeemmedi1 is not an outTransitionGlyph in any other lookup.

Suppose there is another lookup L2 in which jeemmedi1 is an outTransitionGlyph:

Lookup L2:
Jeemmedi1 → jeemmedi6
Whenever it is followed by seenfina

In this case lookup L1 becomes a dependent lookup because it is dependent on the position of L2. Consider the following input from right to left:

seenfina jeemmedi1 bayinint1

If L2 is placed before L1, then L1 will not get processed because it does not get the context jeemedi1 which has already been replaced to jeemedi6 after the application of L2.

	seenfina jeemmedi1 bayinint1
after L2:	seenfina jeemmedi6 bayinint1
after L1:	No change
Output:	**seenfina jeemmedi6 bayinint1**

If on the other hand L2 is placed *after* L1 then L1 gets processed and the result is different. Again using seenfina jeemmedi1 bayinint1 as input

	seenfina jeemmedi1 bayinint1
after L1:	seenfina jeemmedi1 bayinint3
after L1:	seenfina jeemmedi6 bayinint3
Output:	**seenfina jeemmedi6 bayinint3**

Hence L1 is also a dependent lookup (which is *lookup configuration 1*).

Sometimes a lookup may have both default and non-default outTransitionGlyph glyph(s) and can be considered as a dependent *Lookup configuration 3 or 4* lookups. Also a lookup may have both default and non-default context glyph(s). In such case, given that the outTransitionGlyph glyphs(s) are all default shapes, the lookup can be classified as independent only if the context contains the complete domain of shapes the context glyph can acquire. This is illustrated with another example that requires two additional lookup L3 and L4.

Lookup L3:

bayinit1 → bayinit3

whenever it is followed by jeemfina or

whenever it is followed by jeemmedi1

whenever it is followed by jeemmedi2

….   Jeemmedin

Lookup L4:

Jeemmedi1 → jeemmedi2

Whenever it is followed by seenfina

Algorithmically, L3 should be dependent on L4 because L3 has as context jeemmedi2, which is realized only after the application of L4. Even with such inference, it can be seen that L3 is *independent* of lookup L4 because the context contains the complete domain or range of shapes the context glyph i.e. jeem can acquire. As a result L3 will get processed in any case irrespective of whether L4 is placed / positioned before or after L3.

In conclusion, a lookup whose outcome is not affected by its position in the lookup list is classified as independent lookup whereas lookups that yields different output by simply shifting its position in the lookup list are dependent lookups.

11.2. Lookup Arrangement

After the lookups have been categorized then the OpenType font is modified in such a way that all independent lookups are placed before the dependent ones. The font designer should specify through some variable in the OpenType specification the number of independent lookups.

11.3. The Working

Assuming the number of independent lookups is n then the algorithm works as follows

```
Starting from element IND = 1 to length (input string)
    Directly apply all independent lookups that have a rule for string [IND]
```

As an example suppose the first element in the input string has rules in lookup 2,4,22 and n = 7; Then apply lookup 2 on first element, if not successful apply the fourth lookup. Lookup 22 will not be applied because it is not an independent lookup. Then the dependent lookups are processed according to the newly proposed algorithm.

```
Starting from lookup LK = n to lookupcount
    Starting from element IND = 1 to length (input string)
        if(lookup LK has a rule for element at index IND)
            Apply lookup LK on INDth  index of input string
```

One thing that can be inferred is that more the independent lookups better the efficiency.

Chapter 12

Results

The efficiency of the new algorithm can best be explained in terms of reduction in processing time. A simple methodology was adopted to calculate the overall processing time of the algorithm on given input. For a given input, the time immediately before the algorithm was noted and stored. Immediately after the algorithm completes execution, the time is again noted. The difference indicates the real-time running time of the algorithm.

12.1. Time Difference

The time for the algorithm before and after modification was calculated and tabulated. Three samples of time (in seconds) for each 3, 10, 25, 40, 60, 120 page documents written in Nafees Nastaliq were obtained and are given next.

12.1.1 Old Method

The table below gives the time in seconds for the original algorithm.

No. of samples / No. of Pages	1	2	3
3	1.16	1.19	1.18
10	3.67	3.69	3.71
25	10.08	10.14	10.27
40	15.29	15.49	15.38
60	26.96	27.49	26.97
120	46.11	46.01	46.2

Table 5 Time for original Algorithm

12.1.2 The Efficient Glyph Processing Algorithm

The next table shows the results for the algorithm that uses the glyph_lookup_map table for selective processing of lookups.

No. of Pages \ No. of samples	1	2	3
3	0.23	0.24	0.22
10	0.75	0.74	0.77
25	1.99	1.96	1.99
40	3.01	3.12	3.07
60	5.67	5.52	5.24
120	9.22	9.11	9.23

Table 6 Time for modified Algorithm

12.1.3 Further Enhancement

The next table shows the results for the algorithm that uses the glyph_lookup_map table along with the dependent independent information for selective processing of lookups.

No. of Pages \ No. of samples	1	2	3
3	0.20	0.21	0.21
10	0.70	0.69	0.69
25	1.92	1.93	1.90
40	3.03	3.03	3.04
60	5.60	5.61	5.62
120	9.19	9.18	9.20

Table 7 Time for Dependent/Independent Lookup Algorithm

12.2. The t-Test

The collected samples were analyzed using the *t*-Test method. The *t*-Test is a statistical method that is used to solve problems associated with inference based on "small" samples: the calculated mean (*X*avg) and standard deviation (σ) may by chance deviate from the "real" mean and standard deviation. In this test the samples (group A) for 3-page document using the old algorithm as given in table 5 were compared with the samples (group B) using the modified algorithm as given in table 6. The details of the test are given on table 8 below.

Group A: Number of items= 3 1.16 1.18 1.19	Group B: Number of items= 3 0.220 0.230 0.240

Mean = 1.18
95% confidence interval for Mean: 1.156 to 1.197
Standard Deviation = 1.528E-02
Hi = 1.19 Low = 1.16
Median = 1.18
Avg. Absolute Dev. from Median = 1.000E-02
	Mean = 0.230
	95% confidence interval for Mean: 0.2093 thru 0.2507
	Standard Deviation = 1.000E-02
	Hi = 0.240 Low = 0.220
	Median = 0.230
	Avg. Absolute Deviation from Median = 6.667E-03

t = 89.8 Standard Deviation = 0.129E-01 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	

Table 8 t-Test for 3-Page samples given in table 5 and Table 6

Similarly *t*-test was carried out for the samples (group B) for 3-page document using the first modified algorithm as given in table 6 and the samples (group C) using the dependent/Independent lookup algorithm as given in table 7. Table 9 summarizes this test.

Group B: Number of items= 3 0.220 0.230 0.240	Group C: Number of items= 3 0.200 0.210 0.210

Mean = 0.230
95% confidence interval for Mean: 0.2093 to 0.2507
Standard Deviation = 1.000E-02
Hi = 0.240 Low = 0.220
Median = 0.230
Avg. Absolute Deviation from Median = 6.667E-03
	Mean = 0.207
	95% confidence interval for Mean: 0.1936 thru 0.2198
	Standard Deviation = 5.774E-03
	Hi = 0.210 Low = 0.200
	Median = 0.210
	Avg. Absolute Deviation from Median = 3.333E-03

t= 3.50 Standard Deviation = 0.816E-02 degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.025 reducing the possibility that the measured difference between the two samples is most likely not due to chance.	

Figure 9 t-Test for 3-Page samples given in table 6 and Table 7

As can be seen from the tables above the dependent/independent information used for selective lookup processing yields best results. The output of this algorithm is greatly dependent on the number of independent lookups. If a font has no independent lookups, then this algorithm is no better than the previously recommended method. Moreover, this puts the extra burden on the font developer to correctly specify the independent and dependent lookups and position or order them correctly.

References

[1] http://www.microsoft.com/typography/default.asp

[2] OpenType Specification v1.2, available at www.microsoft.com/typography/tt/tt.htm

[3] http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&cat_id=RenderingGraphite

[4] www.Freetype.org

[5] www.unicode.org

[6] Hudson John, *"Unicode from Text to type ",*Language Culture Type: International Type Design in the Age of Unicode, Typographique International, NY 2002.

[7] http://www.microsoft.com/truetype/tt/tt.htm

[8] http://www.microsoft.com/typography/otfntdev/arabicot/default.htm: "Creating and supporting OpenType fonts for the Arabic script" Microsoft 2001.

[9] www.Redhat.org

[10] Phinney Thomas, *"TrueType, PostScript Type 1 & OpenType: What's the difference?"* December 2002.

[11] Taylor Owen, *"Pango: Internationalized text handling"* lwn.net/2001/features/OLS/pdf/pdf/pango.pdf

[12] Microsoft, 1992, *"TrueType Font Technology, An Overview of its Implementation in Microsoft Windows Version 3.1"*, Microsoft Corporation.

[13] Jenkins John, "*The Unicode Character-Glyph Model: Case Studies",* www.apple.org

Appendix A: Details of t-test

Group A: Time with old Algorithm

Group B: Time with new Algorithm

10-Page Document

Group A: Number of items= 3 3.67 3.69 3.71	Group B: Number of items= 3 0.75 0.74 0.77

Mean = 3.69
95% confidence interval for Mean: 3.661 to 3.791
Standard Deviation = 1.528E-02
Hi = 3.71 Low = 3.67
Median = 3.69
Avg. Absolute Dev. from Median = 1.330E-02
	Mean = 0.753
	95% confidence interval for Mean: 0.7243 thru 0.7810
	Standard Deviation = 1.000E-02
	Hi = 0.770 Low = 0.740
	Median = 0.750
	Avg. Absolute Deviation from Median = 6.667E-03

t = 144 Standard Deviation = 0.129E-01 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	

25-Page Document

Group A: Number of items= 3 10.08 10.14 10.27	Group B: Number of items= 3 1.99 1.96 1.99

Mean = 10.2
95% confidence interval for Mean: 10.05 thru 10.28
Standard Deviation = 9.713E-02
Hi = 10.3 Low = 10.1
Median = 10.1
Avg.A bsolute Deviation from Median = 6.333E-02
	Mean = 1.98
	95% confidence interval for Mean: 1.868 thru 2.092
	Standard Deviation = 1.732E-02
	Hi = 1.99 Low = 1.96
	Median = 1.99
	Avg. Absolute Deviation from Median = 1.000E-02

t = 202 Standard Deviation = 0.178E-01 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	

40-Page Document

Group A: Number of items= 3 15.29 15.49 15.38	Group B: Number of items= 3 3.01 3.12 3.07

Mean = 15.4
95% confidence interval for Mean: 15.26 thru 15.52
Standard Deviation = 0.100
Hi = 15.5 Low = 15.3
Median = 15.4
Avg. Absolute Deviation from Median = 6.667E-02
	Mean = 3.07
	95% confidence interval for Mean: 2.937 thru 3.196
	Standard Deviation = 5.508E-02
	Hi = 3.12 Low = 3.01
	Median = 3.07
	Avg. Absolute Deviation from Median = 3.667E-02

t = 187 Standard Deviation = 0.80E-01 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	

60-Page Document

Group A: Number of items= 3 26.96 27.49 26.97	Group B: Number of items= 3 5.67 5.52 5.24

Mean = 27.1
95% confidence interval for Mean: 26.72 thru 27.56
Standard Deviation = 0.303
Hi = 27.5 Low = 27.0
Median = 27.0
Average Absolute Deviation from Median = 0.177
	Mean = 5.48
	95% confidence interval for Mean: 5.053 thru 5.900
	Standard Deviation = 0.218
	Hi = 5.67 Low = 5.24
	Median = 5.52
	Average Absolute Deviation from Median = 0.143

t = 100 Standard Deviation = 0.264 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	

120-Page Document

Group A: Number of items= 3 1.16 1.18 1.19	Group B: Number of items= 3 9.22 9.11 9.23

Mean = 31.6
95% confidence interval for Mean: 30.96 thru 32.23
Standard Deviation = 0.542
Hi = 32.2 Low = 31.2
Median = 31.4
Average Absolute Deviation from Median = 0.343
	Mean = 6.27
	95% confidence interval for Mean: 5.641 thru 6.906
	Standard Deviation = 0.130
	Hi = 6.40 Low = 6.14
	Median = 6.28
	Avg. Absolute Deviation from Median = 8.667E-02

t = 78.6 Standard Deviation = 0.394 Degrees of freedom = 4 The probability of this result, assuming the null hypothesis, is 0.000 indicating that measured difference between the two samples is most likely not due to chance.	