Progress on the Omega Project

John Plaice
Département d'informatique, Université Laval, Ste-Foy, Québec, Canada G1K 7P4
plaice@ift.ulaval.ca

Abstract

Omega (Ω) consists of a series of extensions to TEX that improve its multilingual capabilities. It allows multiple input and output character sets, and will allow any number of internal encodings. Finite state automata can be defined, using a `flex`-like syntax, to pass from one coding to another. When these facilities are combined with large (16-bit) virtual fonts, even scripts that require very complex contextual analysis, such as Arabic and Khmer, can be handled elegantly.

A year ago (Plaice, 1993), a proposal was made to add the notion of *character cluster* to TEX, and that in fact this notion would be included in an extension of TEX called Ω. The fundamental idea would be that any sequence of letters could be clustered to form a single entity, which could in turn be treated just like a single character. Last year's proposal was not accompanied with an implementation. That is no longer the case, and so the notion of character cluster is now much clearer. Essentially, the input stream passes through a series of filters (as many as are needed), and all sorts of transformations become possible; for example, to handle different character sets, to do transliterations or to simplify ligature mechanisms in fonts. In addition, TEX's restrictions to eight-bit characters have been eliminated.

Encodings and recodings

If we abstract ourselves from the problems associated with layout, typesetting can be perceived as a process of converting a stream of characters into a stream of glyphs. This process can be straightforward or very complex. Probably the simplest case is English where, in most cases, the input encoding and the font encoding are both ASCII; here, no conversion whatsoever need take place. At the other extreme, we might imagine a Latin transcription of Arabic that is to generate highly ligatured, fully vowelized Arabic text; here, the transliteration must be interpreted, the appropriate form of each consonant selected, then the ligatures and vowels chosen — the process is much more complex.

TEX supposes that there are two basic encodings: the input encoding and the internal encoding, each of which uses a maximum of eight bits. The conversion from the input encoding to the internal encoding takes place through an array lookup (`xord`). An input character is read and converted according to the `xord` array. The font encoding is the same as the internal encoding, except of course for the fact that several characters can combine to form ligatures.

Suppose that one works in a heterogeneous environment and that one regularly receives files using several different encodings. In this case, one is faced with a problem, because the TEX conversion of input to internal encoding is hard-wired into the code. To change the input encoding, one actually has to change TEX's code — hardly an acceptable situation.

So how does one get around this problem? The first possibility is to use preprocessors, which might themselves be faulty, before actually calling TEX. The second is to use active characters: at the top of every file, certain characters are defined to be macros. However, this process is unreliable, since other macros might expect those characters to be ordinary letters.

Much more appropriate would be to have a command that states that the input encoding has changed and, on the fly, that TEX switches conversion process, maintaining the same internal coding (if we are still in the same document, we probably want to use the same font).

It would probably not be too much trouble to adapt TEX so that it could quickly switch from one one-octet character encoding to another one. However, there are now several multi-octet character sets: JIS, Shift-JIS and EUC in Japan, GB in China and KSC in Korea. Some of these are fixed-width, stateless 16-bit codes, while others are variable-width codes with state. Also, now that the base plane of ISO-10646-1.2 (Unicode-1.1) has been defined, we have a 16-bit character set that can be used for most of the world's languages. However, for reasons of compatibility, we may often come across files in UTF format, where up to 32 bits can be stored in a variable width (1-6 octets) encoding, but for which ASCII bytes remain ASCII bytes. In other words, the conversion process from input to internal encoding is not at all simple.

To complicate matters even further, it is not at all clear what the internal encoding should be. Should it be fixed, in which case the only reasonable possibility is ISO-10646-1.2? Or should the internal coding itself be variable? If the internal coding is fixed, that will mean that in most cases a conversion from internal encoding to font encoding will have to take place as well. For example, few Japanese fonts are internally encoded according to Han Unification, the principle behind ISO-10646-1.2. Rather, the internal encoding would be by Kuten numbers or by one of the JIS encodings. If that is also the case for the input encoding, then a double conversion, not always simple, nor necessary, would have to take place.

To make matters even worse, one's editor may not always have the right fonts for a particular language. Transliteration becomes a necessity. But transliteration is completely independent from character encodings; the same Latin transliteration for Cyrillic can be used if one is using ISO-646 or ISO-10646. Nor does transliteration have anything to do with font encodings. After all, one would want to use the same Arabic fonts, whether one is typing using a Latin transliteration in ISO-8859-1, or straight Arabic in ISO-8859-6 or ISO-10646.

And, to finish us off, the order of characters in a stream of input may not correspond to the order in which characters are to be put on paper or a screen. For example, as Haralambous (1993) has explained, many Khmer vowels are split in two: one part is placed to the left of a consonantal cluster, and the other part is placed to the right. He has faced similar problems with Sinhalese (Haralambous 1994).

Finally, we should remember that error and log messages must also be generated, and these may not necessarily be in the same character set as either the input encoding or the internal encoding.

Transliteration and contextual analysis. It seems clear that the only viable internal encoding is the font encoding. However, there is no reason that the conversion from input encoding to internal encoding should take but one step. Clearly one can always do this, and in fact, if our fonts are sufficiently large, we can always do all analysis at the ligature level in the font. However, such a decision prevents us from separating distinct tasks, such as — say, for Arabic — first converting all text to ISO-10646, then transliterating, then computing the appropriate form of each letter, and only then having the font's ligature mechanism take over.

In fact, what we propose is to allow any number of filters to be written, and that the output from one filter can become the input to another filter, much like UNIX pipes.

Ω Translation Processes

In Ω, these filters are called Translation Processes (ΩTPs). Each ΩTP is defined by the user in an `.otp` file: with a syntax reminiscent of the Flex lexical analyzer generator, users can define finite state Mealy automata to transform character streams into other character streams.

These user-defined translations tables are not directly read by Ω. Rather, compact representations (`.ctp` files) are generated by the `OTPtoCTP` program. A `.ctp` file is read using the Ω primitive `\otp` (see below). Here is the syntax for a translation file:

```
in:           n;
out:          n;
tables:       T*
states:       S*
aliases:      A*
expressions:  E*
```

where n means any number. Numbers can be either decimal numbers, WEB octal (`@'...`) or hexadecimal (`@"...`) numbers, or visible ISO-646 characters enclosed between a grave accent and a single quote (`` `c' ``).

The first (second) number specifies the number of octets in an input (output) character (the default for both is 1). These numbers are necessary to specify the translation processes that must take place when converting to or from character sets that use more than one octet per character.

Tables are regularly used in character set conversions, when algorithmic approaches cannot be simply expressed. The syntax for a table T is:

id`[`n`] = {`$n, n, \ldots, n$`};`

The ΩTPs, as in Flex, allow a number of states. Each expression is only valid in a given state. The user can specify when to change states. States are often used for contextual analysis. The syntax for a set S of states is:

id, *id*, …, *id*;

Expressions are pattern-action pairs. Patterns are written as simple regular expressions, which can be aliased. The syntax for an alias A is:

id = L;

where L is a pattern.

If only one state is used, then an expression E consists of a pattern and an action:

L => R* ;

where the syntax for patterns is:

```
L  ::=  n
     |  n-n              range
     |  .                wildcard
     |  LL               concat.
     |  L{n,m}           occurrences
     |  (L | ... | L)    choice
     |  ^(L | ... | L)   negative choice
     |  {id}             abbreviation;
```

and where the simplified syntax for actions is:

```
R  ::=  string
     |  n
     |  \n
     |  \($ − n)
     |  \(* + n − n)
     |  #(R)
     |  id[R]
     |  R op R          arithmetic;
```

Patterns are applied to the input stream. When a pattern has matched the input stream, the action to the right is executed. A *string* is simply put on the output stream. The \\$n$ refers to the n-th matched character and the \\$ refers to the last matched character. The \\$*$ refers to the complete matched substream, while \\($*-n$) refers to all but the last n characters. Table lookup is done using square brackets. All computations must be preceded by a #.

Here is a sample translation from the Chinese GB2312-80 encoding to ISO-10646:

```
in:  1;
out: 2;
tables: tabgb[8795] = {...};
expressions:
(@"00-@"A0)              => \1;
(@"A1-@"FF)(@"A1-@"FF)   =>
     #(tabgb[(\1-@"A0)*@"64 + (\2-@"A0)]);
.                        => @"FFFF;
```

where we use @"FFFF as the error character. And here is a common transliteration in Indic scripts:

```
{consonant}{1,6} {vowel}  => \$ \(*-1);
```

The vowel at the end is placed before the stream of consonants.

The complete syntax for expressions is more complicated, as there can be several processing states. In addition, it is possible to push values back onto the input stack. Here is the complete syntax:

<state> L => R <= R* <newstate>*

The *state* means that if the ΩTP is in that state then this pattern-action pair can possibly be used. The *newstate* designates the new state if this pattern-action pair is chosen.

Here is an example from the contextual analysis of Arabic:

```
<MEDIAL>{QUADRIFORM}{NOT_ARABIC_OR_UNI}
         => #(\1 + @"DD00)
         <= \2
         <pop:>
         ;
```

When in state MEDIAL (in the middle of a word), a letter with four possible forms is followed by a non-Arabic letter, then the output is the quadriform letter plus the value @"DD00. The non-Arabic letter is placed back on the input stack. Then the current state is popped and the ΩTP returns to the previous state, whatever it was.

Loading ΩTPs. Loading an ΩTP is similar to loading a font. The instruction is simply:

\otp\newname = *filename*

The .ctp file *filename*.ctp is read in and stored in the otp info memory, similar to the font info memory. A number is assigned to the control sequence \newname, as for fonts. Thereafter, one can refer to that ΩTP either through the generated number or through the newly-defined control sequence.

Input encodings. When reading a file from an unknown source, using an unknown character set, some sort of mechanism is necessary to determine what the character set is. There are two possibilities: either use a default character set or have some way of quickly recognizing what the character set is.

Fortunately, most character sets contain ISO-646 as a subset. The ISO-10646-1.2 character set, in both its 16- and 32-bit versions, retains ISO-646 as its original 128 characters. The only widely-used character set that does not fit this mold is IBM's EBCDIC.

We therefore provide the means for automatically detecting the character set family. It suffices that the user place a comment at the very beginning of each file: the % character is sufficient to distinguish each of the families. A file using an 8-bit extension of ISO-646 begins with the character code 0x25; a file with 16-bit characters begins with 0x00 0x25.[1] Finally, a file using the EBCDIC encoding begins with 0x6C. Should there be no comment character, then the default input encoding (ISO-646) is assumed.

Once Ω knows how to read the basic Latin letters, it is possible to *declare* what translation the input must undergo. This is done with the command \InputTranslation, e.g. \InputTranslation 1 states that the entire input stream, starting immediately *after* the newline at the end of this line, will pass through the first ΩTP process.[2]

It is also possible to change the character set within a file. This process is more difficult, as it is not always clear where *exactly* the change is to take place. Suppose that we pass from an 8-bit character set to a 16-bit character set. It is important that we know what the *last* 8-bit character is and what the *first* sixteen-bit character is.

This question can be resolved by specifying a particular character as being the one which changes.

[1] A file with 32-bit characters would begin with 0x00 0x00 0x00 0x25, but the current version of Ω does not support 32-bit characters.

[2] The syntax for the new primitives has not been finalized. In particular, it is not clear that the explicit numbering of filters and translation processes is simple to manipulate. Those who wish to use Ω should check the manual for the exact syntax.

However, to simplify matters, we assume that all input translation changes take place *immediately after* the newline at the end of the line in which the `\InputTranslation` command appears.

Transliteration. Once characters have been read, most likely to some universal character set such as ISO-10646, then contextual analysis can take place, independently of the original character set. This analysis might require several filters, each of which is similar to the translation process undergone by the input.

Since the number of filters that we might want to use is arbitrarily large, there are two commands to specify filters:

`\NumberInputFilters` n

states that the first n input filters are active. The output from the i-th filter becomes the input for the $i+1$-th filter, for $i<n$.

`\InputFilter` $m\ i$

states that the m-th input filter is the i-th ΩTP.

Sequences of characters with character codes 5, 10, 11 and 12 successively pass through the translation processes n translation processes. It should be understood that the result of the last translation process should be the font encoding itself; it is in this encoding that the hyphenation algorithm is applied.

Our Arabic example then looks like this:

```
\otp\trans     = ISO646toISO10646
\otp\translit  = TeXArabicToUnicode
\otp\fourform  = UnicodeToContUnicode
\otp\genoutput = ContUnicodeToTeXArabicOut
\InputFilter 0 \translit
\InputFilter 1 \fourform
\InputFilter 2 \genoutput
\NumberInputFilters 3
```

The `TeXArabicToUnicode` translator takes the Latin transliteration and converts it into Arabic. As for `UnicodeToContUnicode`, it does the contextual analysis for Arabic; that is, it takes Arabic (in ISO-10646) and, using a private area, determines which of the four forms (isolated, initial, medial or final) each consonant should take. Finally, `ContUnicodeToTeXArabicOut` determines what slot in the font corresponds to each character. Of course, nothing prevents the font from having its own sophisticated ligature mechanism as well.

Output and special encodings. TEX does not just generate `.dvi` files. It also generates `.aux`, `.log` and many other files, which may in turn be read by TEX again. It is important that the output mechanism be as general as the input mechanism. For this, we introduce the analogous operations:

```
\OutputTranslation
\OutputFilter
\NumberOutputFilters
```

with, of course, the appropriate arguments.

Similarly, in its `.dvi` files TEX can output commands that are device-driver specific, using `\special` commands. Since the arguments to `\special` are themselves strings, it seems appropriate to also allow the following commands:

```
\SpecialTranslation
\SpecialFilter
\NumberSpecialFilters
```

Large fonts

TEX is limited to fonts that have a maximum of 256 characters. However, on numerous occasions, a need has been shown for larger fonts. Obviously, for languages using ideograms, 256 characters is clearly not sufficient. However, the same holds true for alphabetic scripts such as Latin, Greek and Cyrillic; for each of these, ISO-10646-1.2 defines more than 256 pre-composed characters. However, many of these characters are basic character-diacritic mark combinations, and so the actual number of basic glyphs is quite reduced. In fact, for each of these three alphabets, a single 256-character font will suffice for the basic glyphs.

We have therefore decided, as a first step, to offer the means for large (16-bit) virtual fonts, whose basic glyphs will reside in 8-bit real fonts. This is clearly only a first step, but it has the advantage of allowing large fonts, complete with ligature mechanisms, without insisting that all device drivers be rewritten.

In addition to changing TEX, we must also change `DVIcopy` and `VPtoVF`, which respectively become `XDVIcopy` and `XVPtoXVF`. The `.tfm`, `.vp` and `.vf` files are replaced by `.xfm`, `.xvp` and `.xvf` files, respectively. Of course, the new programs can continue to read the old files.

`.xfm` files. The `.xfm` files are similar to `.tfm` files, except that most quantities use 16 or 32 bits. Essentially, most quantities have doubled in size. The header consists of 13 four-octet words. To distinguish `.tfm` and `.xfm` files, the first four octets are always 0 (zero). The next eleven words are the values for *lf*, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, and *np*; all of these values must be non-negative and less than 2^{31}. Now, each *char_info* value is defined as follows:

width index	16 bits
height index	8 bits
depth index	8 bits
italic index	14 bits
tag	2 bits
remainder	16 bits

Each *lig_kern_command* is of the form:

op byte	16 bits
skip byte	16 bits
next char	16 bits
remainder	16 bits

Finally, extensible recipes take double the room.

`.xvp` files. The `.xvp` files are simply `.vpl` files in which all restrictions to 8-bit characters have been removed. Otherwise, everything else is identical.

Minor changes. Since the changes above required carefully examining all of the code for TEX, we took advantage of the opportunity to remove all restrictions to a single octet. So, for example, more than 256 registers (of each kind) can be used. Similarly, more than 256 fonts can be active simultaneously.

Conclusions

The tranformation of TEX into Ω was a necessary step for the development of a typesetting tool that could be used for most (all?) of the world's languages. Scripts that, for various historical and political reasons, retained their calligraphic traditions, can now be printed with ease without sacrificing on aesthetics. In fact, as presented in Haralambous and Plaice (1994), it is now possible to use calligraphic-style fonts for Latin-alphabet languages, without any extra overhead: just change the font and the translation process, everything else is automatic.

Large fonts are definitely useful: all the interactions of characters in a font can be examined. However, it is not necessary to change all our device drivers. A large virtual font might still only reference small real fonts (unlikely to be the case in Eastern Asia, where all fonts are large).

Large fonts, with full interaction between the characters, mean that one can envisage variable-width Han characters. According to Lunde (1993), this topic has been mentioned in several Asian countries.

Finally, I should like to state that the change from TEX to Ω is really quite small. Apart from the idea of character cluster, everything is already there in TEX. It should be considered a tribute to Donald Knuth that so little time was required to make these changes.

Acknowledgements

The Ω project was devised by Yannis Haralambous and myself. It would never have gotten off the ground if it had not been for the numerous discussions that I had with Yannis. Many thanks as well for the discussions in the Technical Working Group on Multiple Language Coordination.

Bibliography

Haralambous, Yannis, "The Khmer script tamed by the lion (of TEX)", *TUGboat* **14**(3), pages 260–270, 1993.

Haralambous, Yannis, "Indic TEX preprocessor: Sinhalese TEX", TUG94 Proceedings, 1994.

Haralambous, Yannis, and John Plaice, "First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer, Unicode", TUG94 Proceedings, 1994.

Lunde, Ken, *Understanding Japanese Information Processing*, O'Reilly and Associates, Sebastopol (CA), USA, 1993.

Pike, Rob, and Ken Thompson, `tcs` program, `ftp://research.att.com/dist/tcs.shar.Z`, 1993.

Plaice, John, "Language-dependent ligatures", *TUGboat* **14**(3), pages 271–274, 1993.