Open-belly surgery in Ω_2

YANNIS HARALAMBOUS, GÁBOR BELLA, ATIF GULZAR
ENST Bretagne, CS 83 818, 29 283 Brest Cedex 3, France
`yannis dot haralambous (at) enst-bretagne dot fr, gabor dot bella (at) enst-bretagne dot fr, atif dot gulzar (at) gmail dot com`

Abstract

TEX and its successors, including the initial version of Ω, all suffer from the same technical limitations, such as inadequate support for TrueType/OpenType font formats and the lack of distinction between character and glyph data. In this paper, the authors present Ω_2, which provides extensibility through both external modules *and the* texteme concept *that supersedes TEX's tokens and nodes as well as characters and glyphs. Ω_2's modules, while much more powerful than macros or ΩTPs, provide relatively easy access to Ω_2's internals without needing to touch the source code itself. Among immediate applications are full OpenType support (GSUB, GPOS, etc.), use of independent linguistic tools such as hyphenation algorithms, and support for Unicode's Bidirectional Algorithm.*

Introduction

Since its birth, TEX has undergone significant evolution, resulting in extended versions such as ε-TEX, pdfTEX, Ω_1, and others. However, the fundamentals of TEX have barely changed: to cite two examples, both its basic text model, that is, the *horizontal node list*, and the concept of the *single main vertical list* are almost exactly the same as 25 years ago. Ω_1, as a first step towards Unicode compatibility, introduced 16-bit character codes and some text directionality support but did not change TEX's original text model, based on token lists converted to node lists and finally to DVI instructions.

Users and developers have long since recognised the serious limitations of this approach. Inside the belly of TEX, character codes included in tokens are replaced by glyph codes, resulting in loss of information if one does not stick to the severely limited TEX font encodings, especially in the case of non-Latin scripts: searchability and recovery of the original character data in general become impossible. Support for advanced font formats such as OpenType, essential for writing systems having contextual properties (Arabic, Hebrew, Nastaleeq, the Indic scripts, etc.) but also necessary for some Western typographical features, is also impossible without a clear distinction between the concepts of character and glyph. Still due to the same limitation, until now, no successor of TEX could get rid of the TFM font format and provide native support for PostScript or TrueType-based fonts.

As of today, the most remarkable development in the TEX world regarding support for intelligent font formats is without doubt the XƎTEX system [1]. However, as far as micro-typography is concerned, XƎTEX does not have much to do with Knuth's original TEX: while the latter is a stand-alone tool, XƎTEX 'outsources' all the word-level typography to the underlying operating system and external libraries: the ICU library initially developed by IBM [7], ATSUI under Mac OS X, FreeType under Linux, all come into play. So, while XƎTEX succeeds in combining the OpenType and Apple AAT font technologies with TEX's layout and input style, it ties the application to the operating system.

The main reason for preferring such a solution was without any doubt the opportunity to avoid reimplementing the quite complicated Unicode and OpenType engines that already exist on the operating system level. Moreover, the TEX source code itself is far from being easily extendable, despite having been written in the didactic WEB programming language that divides the code into small, easy-to-digest chunks. It lacks modularity and is so highly optimised for performance that the slightest modification can cause a snowball effect of patching and debugging. The single way of extending TEX foreseen by Knuth was the creation of new *whatsit* node types, each of which in practice results in a further increase of the programme's complexity.

Unlike XƎTEX's approach, the developers of Ω_2 preferred to solve the problem of extensibility by introducing modularity into the system. In fact, ΩTPs were already module-like components in Ω_1. ΩTPs are capable of transforming 'character' strings into other 'character' strings and even of inserting new control sequences. However, due to the early, token-level stage where they intervene, they are limited by the grouping of input text: to show an example, processing of the word '*emph*asis':

```
{\it emph}asis
```

as a whole is not possible, since it is broken by markup into two separate ΩTP buffers. Even more important is the fact that ΩTPs are inherently character-level tools and are unable to perform operations other than character substitutions (such as glyph positioning, adding linguistic data, modifying colour, etc.).

Consequently, the objectives that the Omega team have set to themselves are on one hand to solve the character/glyph duality issue, by creating data structures capable of storing both, and on the other hand to provide extensibility to Ω_2 in a more efficient way, in order to allow manipulation of characters, glyphs, and other types of data independently. It will be shown later in the article how these two improvements are tightly related and that they provide the best results when used together.

In the following section, the original text model of TEX will be compared to our *texteme*-based approach. Then, external modules will be presented in detail. Finally, it will be shown how using modules together with texteme properties opens up possibilities of immediate applications such as OpenType support, linguistic analysis, or fully customised typography beyond the limitations of TEX or current font technologies.

Of characters, glyphs, tokens, nodes, and DVI instructions

Text in TEX and in its extensions goes through several different states. In the beginning, it is read as *character data* from the input buffer. These characters may either be textual content or TEX markup. They will immediately be converted into either *character tokens* or *control sequence tokens*, respectively. A character token consists of the character code and its catcode, while a control sequence is represented by a single identifier. However, the token state is ephemeral: shortly after their creation, both types of tokens are converted into *nodes*.[1] Character tokens usually become character nodes, but sometimes also ligature nodes. Other types of nodes are also created on-the-fly: kern nodes, glue nodes, discretionary nodes, and so on. Text is organised into horizontal and vertical lists (represented by *hlist* and *vlist* nodes).

An important thing to notice is that fonts come into the picture precisely at the point of converting tokens into nodes. (Ligature, kerning, glue, etc., information all come from font resources.) This is the very moment of TEX's original sin: supposing that

$$\textit{character code} \equiv \textit{glyph code from the font},$$

characters are being *replaced* by glyphs in character nodes. The reason why hyphenation, in principle a character-based operation, still works at a latter stage[2] is this assumed equality, that is in fact valid only for a small set of characters, namely those that were coded in locations common between character and font encodings. Were we to use a different (say, OpenType) font format or a script like Arabic, subsequent character-based operations such as searching, copying and pasting, and hyphenation would all be doomed to failure.

The odyssey is still not over: TEX, having done most of its work on node lists, in the end outputs the resulting document using the venerable DVI format where text is encoded through glyph identifiers only, represented in 16 or 8 bits, depending on whether Ω_1 or another TEX-based system is being used. By this time, all the other types of nodes holding non-character data either have already been absorbed in the typesetting process (penalty, discretionary, etc.), or else they now become physical dimensions (kerning, offsets, glue, etc.) or special DVI instructions (specials, etc.). This is the end of the story: our output DVI document is purely presentation-oriented and in no way is able to provide the original character data, long lost in the process.

Textemes

Textemes[3] are one solution to the problems presented above. The idea is to replace TEX's various data repre-

[1]ΩTPs extend the lives of tokens somewhat: they read character tokens and output both character and control sequence tokens.

[2]Namely, at line breaking.

[3]Introduced as *signs* at the EuroTEX 2005 conference.

sentations, namely characters, character tokens, some types of nodes, as well as glyphs in the output, by a single entity: the texteme.

A texteme, as presented in detail in [5] and in [4], is a *set of properties*, where a property is basically a *key–value pair*. A texteme usually represents a character, its glyph, and other related data. More specifically, character code, glyph and font identifiers, and any kind of information related to the atomic units of electronic text are all represented by texteme properties.

How do textemes work in Ω_2? The general idea is to let information accumulate inside textemes instead of converting data from one form to the other. Raw, unformatted text is a texteme string where textemes contain only *character* properties. When raw text (with markup) is fed into Ω_2, a *catcode* property is added to every texteme. When font information is read, instead of creating character nodes, the same textemes — texteme nodes — are carried on, only with new *glyph* and *font* properties added. No ligature nodes are created: an 'ﬁ' ligature is represented by two textemes linked together, the first with *character*=`f` and *glyph*=ﬁ, and the second with *character*=`i` and *glyph*=∅ (empty). Some other information like kerning, glue, or penalties, become texteme properties just the same, resulting in simplified text structure.

Separating character and glyph codes while having access to both of them throughout the whole typesetting process proves to be very useful for tasks such as hyphenation (as shown in Haralambous's article [3]). However, the ultimate goal is to be able to produce final documents that keep all these accumulated (and useful) information. A document that displays glyphs but also holds the original character-based text has an enormous technical advantage compared to glyph-only documents where retrieval of characters is only possible through non-standard, error-prone glyph naming schemes.

The PDF format makes such a double encoding of text possible through the `ActualText` operator: for every glyph or glyph sequence, the corresponding character or character sequence may be defined. This way, even cases like multiple glyphs corresponding to a single character or reordered glyph sequences can be handled correctly, something that would never be possible through glyph naming.

Unfortunately, Ω_2 does not (yet) produce PDF directly and the DVI format does not offer mechanisms similar to PDF's `ActualText`. It is therefore not possible to output texteme data into DVI without breaking compatibility with the original DVI format. As a temporary solution, Ω_2's DVI format has been slightly modified in order to include texteme-related information that is interpreted by a patched `dvipdfmx` utility that produces PDF documents with `ActualText` operators. This is a quick and dirty solution, but it works.

The document creator is by all means allowed and encouraged to invent and use their own properties in their documents. First of all, the set of available texteme properties is open and extensible.[4] Such user-defined properties can be added either automatically, by linguistic analysers and various text processor tools, or manually, by a texteme-compatible text editor (a simple prototype of which has been developed by students of ENST Bretagne). In this editor, texteme properties are added and manipulated in an intuitive, graphical way. Texteme-based documents can then be saved in XML that Ω_2 will be able to interpret and thus rebuild texteme strings. (The XML reading capability of Ω_2 has not been developed yet.)

How do texteme properties come into play during text processing? External modules are the answer.

External modules

Theoretical considerations

As mentioned before, Ω_1's ΩTPs are basically character processors at the token level. This approach is not sufficient when non-character data needs to be processed (e.g., glyph substitution or glyph positioning). First, with the introduction of textemes, access to individual texteme properties as opposed to mere character strings becomes necessary. Secondly, even if an extended ΩTP syntax and input/output scheme allowed the handling of such information, ΩTPs are still called at the token level where font data have not yet been read by Ω_2.

Consequently, in order to allow ΩTP-like external modules to process font-dependent information, their point of activation needs to be displaced to a later point, to the node level.

But there is a problem: since nodes (including texteme nodes) and node lists are considerably more complicated data structures than characters or tokens, Ω_1's internal ΩTP approach is not powerful enough to de-

[4]Namespaces are used for semantic disambiguation.

scribe transformations on them. For these reasons, our new external modules need to be standalone binaries that communicate with Ω_2 using a well-defined XML format (more on this later). Since these binaries can be written in any programming language, there is no limitation to their computing power, unlike former internal ΩTPs that were equivalent to finite state automata.

In reality, there are no less than three well-defined points during Ω_2's typesetting process where external modules may be called:

1. on token-based input text (as in the case of ΩTPs);
2. on yet unbroken horizontal node lists that represent whole paragraphs;
3. on node lists representing individual lines during paragraph breaking.

Each of these three legal intervention points corresponds to a set of well-defined processing tasks. The first point is used by character-level transformers and analysers. They receive a simple list of textemes, uninterrupted by control sequences on grouping braces (no wonder: these tokens act as boundaries for the ΩTP buffer), and containing mostly character information. They are supposed either to perform character transformations (e.g., converting from a local transcription scheme or encoding to Unicode, preprocessing, etc.) or to generate new texteme properties. (At the moment of writing the article, Ω_2 is not yet capable of adding texteme properties at this stage.)

The second type of external module reads entire paragraphs and operates on node lists. This type of module applies, among others, OpenType glyph substitution and positioning rules. However, as a result of working with nodes instead of just characters, such modules have enormous power as well as responsibility over the behaviour of Ω_2: they have full access to every aspect of the text including horizontal and vertical lists, kerning, penalties, and so on. Were a module to, say, substitute a glyph by another, it would have the responsibility to update the corresponding kerning information or at least make sure that this will be done subsequently either by Ω_2 or by another module.

Finally, the third type of module is called on individual lines, inside the line breaking algorithm. Similarly to modules of the second type, it operates on node lists. Its task is to perform line-related operations such as optical kerning, OpenType JSTF (justification) support, or line-dependent glyph substitutions and positionings (e.g., an OpenType contextual ligature invalidated by a nearby line break).

The fragility of node lists when manipulated externally may seem worrying. Indeed, it is very easy to produce typographically unacceptable documents and even to freeze Ω_2 through erroneous or malicious node operations. Creators of modules should respect rules regarding what and in what order they are allowed to modify. Correct ordering of modules is of crucial importance: for example, the order *character transformations – glyph substitutions – glyph positionings* should always be respected, otherwise regression problems may arise.

Indeed, node-level modules represent a drastic surgical intervention in Ω_2's digestive system: it is as if Ω_2's stomach and intestines were piped into external digestion machines. The reader will kindly excuse the authors for this somewhat disturbing analogy and read on to see how in practice modules are called from Ω_2.

Modules in Practice

Module support in Ω_2 is currently in prototype stage, that is, developer- and user-friendly macros and libraries are only minimally available at the moment.

External modules are implemented as standalone binaries and communicate with Ω_2 through signals and an input-output buffer. For performance reasons, these binaries run as *daemons*, that is, they are spawned only once and then remain idle until they receive data to process as well as a wakeup signal. Once a module has finished processing, after writing its output into the same input-output buffer, it goes back to sleep again, and Ω_2 continues by waking up the following module.

More precisely: a module binary is launched by the `\registermodule` primitive. By writing

```
\registermodule{mymodule}{modbin}{par}{10}
```

the binary programme `modbin` is run, in the future referenced by the name `mymodule`. The `par` parameter means that it is a type 2 (paragraph-level) module and its number in the execution order among modules of the same type is `10`. These four parameters are mandatory. There is a fifth, optional parameter (omitted in our example) that sends its argument to the binary when it is launched; this may sometimes be useful for initialisation purposes.

Modules are *asleep* by default. For performance reasons, Ω_2 does not send any data to sleeping modules. In order to perform their task, modules need to be both *waked up* and *activated*. They are waked up and sent back to sleep by the `\wakeup{mymodule}` and `\gotosleep{mymodule}` primitives. Note that paragraph- and line-level modules are waked up for *entire paragraphs*, there is no point in trying to wake them up for shorter text segments. *Awake* but *inactive* modules receive and read all data but let them pass through untouched. They activate themselves when they encounter an *activate* special node, inserted by the `\activate{mymodule}` macro. From this point, they process the text until they either read a *deactivate* node or arrive at the end of the buffer. These *activate* and *deactivate* nodes may of course very well appear inside paragraphs, making it possible to activate modules for text segments as small as individual characters (more precisely, textemes).

Text (textemes and other nodes) is transmitted between Ω_2 and modules in XML format. The full DTD is not provided here for space reasons; instead, a small but relevant example is given. As is shown, for paragraph and line-level modules, Ω_2's *current font table* is also included in the XML buffer since these modules (e.g., an OpenType engine) usually need access to fonts. The font table is then followed by the node list itself: in our case, a *whatsit*, an empty *horizontal list* and two *texteme* nodes. In this simple example, texteme nodes contain only three properties each, namely the character and the corresponding font and glyph ID.

```
<?xml version="1.0"?>
<buffer version="0.1">
  <preamble>
    <fontlist>
      <font id="51" name="ptmr"
            size="1310720"/>
      <font id="52" name="pala"
            size="655360"/>
      ...
    </fontlist>
  </preamble>
  <nodelist>
    <!-- whatsit -->
    <wha st="6" intpnl="0" brkpnl="0"
         pardir="0">
      <lbl></lbl><lbr></lbr>
    </wha>
    <!-- hlist -->
    <hls wd="1310720" dp="0" ht="0"
         shift_amount="0" gse="0" gsi="0"
         go="0" dir="0">
    </hls>
    <!-- texteme -->
    <t linkl="0" linkr="0">
      <p n="c">76</p> <!-- char: L -->
      <p n="g">12</p> <!-- glyph -->
      <p n="f">52</p> <!-- font -->
    </t>
    <t linkl="0" linkr="0">
      <p n="c">111</p> <!-- char: o -->
      <p n="g">142</p>
      <p n="f">52</p>
    </t>
    ...
  </nodelist>
</buffer>
```

A particular advantage of communicating with modules in XML is that certain tasks can thus be implemented by very simple XSLT transformations that are executed by a generic XSLT driver module. This way, the task of implementing a module is reduced to the complexity of writing XSLT code.

Applications of modules and textemes

OpenType support

For quite a long time, the Holy Grail of typesetting systems has been to implement robust support for the TrueType and OpenType font formats. Development has been slow on all platforms, due to the investment required on a very wide scale (full Unicode support, internationalisation, availability of actual fonts). No wonder that no TEX-based system, apart from XƎTEX, has even come close yet to full OpenType compatibility: without the profound changes in TEX's text model described in earlier sections of the present article, or at least similar improvements, OpenType support is not fully possible.

As has been shown in numerous articles, such as [1], the main difficulty of developing OpenType-compatible applications lies in the complexity of operations described in OpenType's GSUB (glyph substitution) and GPOS (glyph positioning) tables. In order to achieve this, apart from providing an appropriate text model, typesetting applications also need a powerful OpenType engine. Fortunately, a lot of effort has already been made in this direction in the free software community, and thus free and cross-platform OpenType libraries are already available: let us mention the *FreeType* [6] and *M17N* [2] projects that both offer OpenType-related functionalities. The new Ω_2 system makes use of both of these libraries.[5]

Basically, two aspects of the OpenType format need to be taken care of in Ω_2: reading metrics and performing glyph transformations. Most TEX-based systems solve the former issue by falling back to utilities such as `ttf2afm` that convert TrueType metrics into TFM files, the OpenType and the TrueType formats being compatible as far as metrics are concerned. This solution works but is inelegant from the user's point of view since installation and use of TrueType or OpenType fonts require several conversion steps and editing of various configuration files. The authors have thus decided to implement direct access from Ω_2 to OpenType metrics, without any need for OFM or other files. At the moment of writing the article, Ω_2 is already capable of reading TrueType metrics directly from the font.[6]

Glyph transformations, on the other hand, are too big a task to implement inside the monolithic Ω_2 code. Modules are an ideal place for such operations. Both paragraph- and line-level modules are going to be necessary: paragraph-level modules will perform both font-independent (multilingual preprocessing similar to what Microsoft's Uniscribe library does) as well as font-dependent OpenType GSUB and GPOS glyph transformations. Finally, the same OpenType module intervenes once again at the line breaking phase if necessary; also, JSTF support can be implemented at this point.

Hyphenation

TEX's original hyphenation procedure is called in the line breaking phase: at this point, text is converted into lowercase, ligatures are replaced by their original characters, and pattern matching is performed. Consequently, the hyphenation algorithm is an integral part of TEX that is difficult to modify or customise according to the special needs of different languages, apart from language-dependent pattern sets. In Ω_2, textemes and modules allow performing hyphenation externally, in a module, opening up the possibility of using more advanced hyphenation algorithms. The general idea is that the external hyphenation module marks potential hyphenation points in words using texteme properties and at the line breaking stage Ω_2 simply selects from the marked hyphenation points the ones giving the least badness.

See [3] (in this same proceedings volume) for a more detailed description of new hyphenation techniques used in Ω_2.

Getting rid of (some) TEX markup

Through properties, textemes provide a new way of enriching electronic text. In some cases, such properties can substitute for markup that would otherwise serve the same purpose. The interest in doing so lies in the simplification of input text, an important gain both from a technical and a usability point of view. Consider the following example of LATEX code:

```
The \verb#\textcolor# command's purpose
is to colorify text, such as this word
in \textcolor{red}{red}.
```

There are several problems with the above snippet: first of all, it is far from being intuitive. To typeset the '\' character, the user needs to use the `\verb` command, which is one way of escaping the special role of the backslash. Also, there is nothing indicating that the first parameter of the `\textcolor` command is the colour parameter and the second is the text to be coloured: neither a human nor an automatic tool, say a preprocessor, can distinguish them without proper knowledge of the `color` LATEX package. Finally, the use of control sequences and delimiters breaks up text into small chunks causing various problems at later processing stages.

A possible solution is to use texteme properties for colour as well as for escaping. For example, with a *catcode = 12* property the user could mark the backslash as textual content. Of course, more user-friendly property name and values could also be used. The same point can be made for various types of spaces (non-breakable, thin, etc.): instead of using active characters like '~' or control sequences like '`\,`', such information can be encoded as orthogonal properties of the same space character. This solution is also far simpler and more intuitive than using Unicode's various *non-breakable space*, *thin space*, *non-breakable* and *thin space*, etc., characters.

[5] M17N, far from being just a multilingual typesetting engine, is a whole set of libraries aiming at providing complete multilingual support.

[6] Kerning data is not read by Ω_2, as this operation is planned to be implemented inside an external module.

Linguistic tools

One of the advantages of textemes is that the set of available properties is open which, together with modularity, makes it possible to integrate Ω_2 with new text processing applications. For numerous linguistic problems that bear some relation to typography, such an approach can be fruitful: automatically finding word boundaries in Thai or Chinese text or distinguishing dots (for abbreviations) and periods (at ends of sentences) in English typography are all complicated linguistic problems that transcend the limits of typesetting engines. If appropriate, standalone linguistic tools already exist and are able to perform the tasks in question; they can be called as external modules of Ω_2 in order to add linguistic information to the input text in the form of texteme properties. Then, an Ω_2 developer just needs to implement a second, much simpler module that interprets such linguistic properties and takes them into account at the typesetting stage (correct line breaks for Thai, changes in the widths of spaces for English).

Conclusions

The second version of the Ω_2 system has two new aspects: *texteme support* and *modularity*. Although still in a prototype stage, the basic framework for running external modules has been implemented in Ω_2 and the underlying text model has also been adapted to the texteme concept. As an addition, the new Ω_2 outputs a DVI format that contains both characters and glyphs, and with a patched `dvipdfmx` utility this information can be incorporated into PDF documents that as a result will be able to provide the reader both with formatted output and with the *original* character string. On the input side, a texteme-compliant text editor tool has been developed, allowing users to enter texteme properties into input documents. OpenType support is partially available: Ω_2 now reads metrics from OpenType files directly. Work is underway for modular GSUB and GPOS support. The authors are also working on moving TEX's hyphenation algorithm into an external module, resulting in easier implementation of improved hyphenation tools.

Work that still needs to be done includes full capabilities of texteme input and output: texteme-based auxiliary (`.toc` etc.) files and input format, as well as development of various multilingual modules including support for OpenType layout tables. An especially important feature that still needs further development is generation of PDF documents with both character and glyph information added. The already mentioned `dvipdfmx` tool is a likely candidate for such an extension. The authors kindly welcome contribution from the TEX community in these areas.

References

[1] Jonathan Kew: *The Multilingual Lion: TEX Learns to Speak Unicode*. 27th International Unicode Conference. *TUGboat* 26:2, 2005, pp. 115–124.

[2] Nishikimi Mikiko, Handa Kenichi, Takahashi Naoto, Tomura Satoru: *The m17n Library — A General Purpose Multilingual Library for Unix/Linux Applications*. Asian Symposium on Natural Language Processing to Overcome Language Barriers (2004). `http://www.m17n.org`

[3] Yannis Haralambous: *New Hyphenation Techniques in Omega 2*. Proceedings of the EuroTEX 2006 (Debrecen, Hungary) conference. In this volume, pp. 98–103.

[4] Yannis Haralambous, Gábor Bella: *Injecting Information into Atomic Units of Text*. ACM Symposium on Document Engineering 2005.

[5] Yannis Haralambous, Gábor Bella: *Omega Becomes a Sign Processor*. Proceedings of the EuroTEX 2005 (Pont-à-Mousson, France) conference, pp. 99–110.

[6] *The FreeType Project*. `http://www.freetype.org`

[7] *International Components for Unicode*. `http://icu.sourceforge.net/`