Adapting Ω to OpenType Fonts

Anish Mehta
Département Informatique
École Nationale Supérieure des Télécommunications de Bretagne
CS 83 818, 29238 Brest Cédex, France
anish_mca@yahoo.com

Gábor Bella
Gabor.Bella@enst-bretagne.fr

Yannis Haralambous
Yannis.Haralambous@enst-bretagne.fr
http://omega.enstb.org/yannis

Abstract

Nowadays, TeX and its successors still use METAFONT or PostScript Type 1 as their primary font formats, while the "outside world" is moving on to OpenType, a likely candidate for *the* future font standard in computer-based typography. The project we are going to present in this article represents the first step of a long-term plan of adapting the full Ω system to OpenType. As a result of our work, the first version of a new, OpenType-compatible *odvips* is available. This article presents the basic concepts of our solution as well as our further development plans for the project.

Résumé

De nos jours, TeX et ses successeurs utilisent toujours METAFONT ou type 1 comme principaux formats de fonte alors que le «monde réel» se tourne de plus en plus vers OpenType, un candidat très probable pour être *le* format de fonte standard du futur. Le projet présenté dans cet article est la première étape vers une adaptation d'Ω à OpenType. Notre résultat est une nouvelle version d'*odvips*, compatible OpenType. Cet article présente les concepts de base de notre solution ainsi que nos projets de développement futurs.

Introduction

OpenType is a relatively new font format developed jointly by Adobe and Microsoft. The goal of the two companies was to replace the widely used PostScript and TrueType fonts with a single, backward compatible format that also provides better support for international scripts and advanced typography. In fact, OpenType is but a wrapper format that can either embed PostScript (CFF, Compact Font Format) or TrueType fonts, plus some additional tables that provide the extra features.

OpenType's possible advantages or drawbacks put aside, one must realise that, due to the strong technological and economic support of its creators, it is very likely to become a de facto digital font standard. Until now, the market's reaction to this new technology has been rather slow, mainly because of the amount of investment a new font format requires from foundries and application developers, not to mention customers. Nevertheless, newly designed fonts come out more and more often in OpenType format, and for non-Latin scripts OpenType's impact is even greater. So, if the TeX community would like to avoid isolation as far as new high-quality fonts are concerned, it is time to jump on the bandwagon and embrace the format as soon as possible.[1]

Our work aims to provide OpenType support for the Ω typesetting system. Among TeX-based systems, Ω is known for its support of non-Latin scripts such as Arabic, Hebrew or various Indic scripts. Since OpenType is the first format to offer proper typesetting for these scripts, implementing OpenType support in Ω is a logical and desirable step forward.

Design Principles

Ideally, *OpenType compatibility* should mean that OpenType fonts are recognised by every link in the document production chain, from the typesetting phase right up to printing. As of today, Ω, just like all TeX-based systems, needs to load a plethora of files at each step of the typesetting process: font metrics and kerning from OFM and OVF files, script-specific typographic features from ΩTP's (Ω Translation Processes), and finally the glyphs themselves from the actual font file, either in PostScript or bitmap format. With OpenType fonts, this would not be necessary: all information mentioned above could be directly retrieved from the OpenType font file itself. The virtual font concept could also be eliminated, as both

1. We are not the first to do this: pdfTeX supports OpenType.

Ω and OpenType are Unicode-compatible. Finally, the same OpenType font could be integrated, partially or in whole, into the final document to be sent to the printer.

However elegant and visionary (not to mention utopian) this scenario might be, it would certainly require a major rewrite of the whole Ω system, ΩTP handling included, as well as of the *odvips* utility. Support of OpenType's advanced typographic tables (GPOS, GSUB, etc.) as well as access to metric information would have to be implemented in Ω internally, thereby short-circuiting OFM and OVF files. Such a solution, however, would mean abandoning Knuth's basic approach, which separates metrics (information needed for text layout) from glyph shapes (only needed at the final step of typesetting). Although this approach was partly inspired by limitations of memory and processing power at the time TEX was designed, it could be argued that the TFM/OFM concept has the advantage of acting as a common interface to the layout engine, independently of the actual font technology used. It also has to be added that extraction of TEX- and Ω-compatible metric information from OpenType tables is far from a trivial operation, as it will be shown later in this article.

Ω's input processing will also need to be extended so that users can turn on or off certain OpenType features by hand. Most importantly, user control is crucial over discretionary features like glyph alternates.

The real problem, however, comes at the final processing step: the sad fact is that there are virtually no printers today with OpenType support built in. Whatever our document format, the OpenType fonts it uses will necessarily have to be converted into a format the printer will recognise, either by the printer driver (available only for Windows and Macintosh) or at the application level. TEX-based systems, and more precisely, *(o)dvips*, produce PostScript documents directly, without using any drivers. Consequently, the only possible solution is to convert OpenType fonts, right before inclusion in the PostScript document, into a common font format recognised by most printers.

In summary, four major areas of the Ω system need changes:

1. extension of Ω's control sequences to allow user control over OpenType features;
2. extraction of metric and kerning information from the OpenType font;
3. support for OpenType's advanced typographic tables, including GPOS, GSUB and BASE;
4. automatic conversion of OpenType fonts to a format supported by a vast majority of today's printers.

Due to the fact that full implementation of these functionalities will require a major rewrite of the whole Ω-*odvips* system, we have decided to approach the problem in three steps. As a response to the needs of the Ω user community, a basic but working solution is going to be published as soon as possible, providing access to OpenType fonts, even if the advanced feature support is far from complete and not as user-friendly as it should be. This transitional solution is still based on OFM and OVF files, and only includes GSUB support. As a second step, in the new Ω_2 system (coming soon to a TEX Live near you) metrics will be read directly from the OpenType font. Finally, full support of OpenType features as well as their corresponding input control sequences will be implemented in the long run.

The rest of the article deals with the improvements we have made so far to provide basic OpenType support. This includes metric extraction, GSUB support and font conversion and subsetting. In order to read the OpenType font and produce the necessary OFM, OTP, etc., files, we have developed a handful of Python utilities, based on Just van Rossum's great *ttx* program and *ttf2pt1* from Mark Heath et al. Also, *odvips* has been patched to use and subset OpenType-originated fonts.

Conversion of Metric and Kerning Information

In its present state, Ω needs font metric files (OFM) in order to typeset glyphs. As long as direct OpenType access from Ω is not possible, the corresponding OFM file will have to be created automatically during installation of the OpenType font. Whatever our input method, conversion of OpenType metric data into Ω-compatible values is not a trivial process: it is not always possible to find an equivalence relation between the two approaches.

Information found in a TFM/OFM file can be divided into the following three categories:

- global information;
- metric information on an individual glyph level;
- information on glyph pairs.

Global information corresponds to font-wide parameters including *checksum*, *design size* of the font, *character coding scheme*, *font family*, *font size*, *font face* and *font dimension* parameters.

Checksum information could be kept as a trace of the original OFM file, but obviously, there is no standard OpenType table to host it.

Design size is usually a reference to the point size at which glyph outlines and spacing display the best. Design size information can be obtained from OpenType's GPOS table via the `size` feature tag, which contains both the design size and a suggested range of sizes.

OpenType fonts use the Unicode encoding to allow handling of large glyph sets. *Font family* information is available in the *naming table* (`name`) with the help of the predefined `nameID` of 1. However, this information is not of much interest as it is not used consistently.

Global information also includes font dimension parameters such as *slant*, *space*, *stretch/shrink*, *xheight* and *extraspace*. Slant in OFM corresponds to the *italicAngle* entry in OpenType's `post` table, which gives the angle in counterclockwise degrees from the vertical, zero for upright text and negative for text leaning right.

The equivalent of TEX's global *space* parameter in OpenType can be taken as the width of the space glyph.

Stretch/shrink information are not available in standard OpenType tables.[2]

Information regarding *xheight* is available as the *sxHeight* field in the OpenType `OS/2` table. If it is not present then we can take the height of the non-ascending lowercase glyph 'x'. This metric specifies the approximate distance between the baseline and the height of non-ascending lowercase letters.

Finally, no equivalent of the *extraspace* parameter exists in standard OpenType tables.

Metric information on an individual glyph level in an OFM file corresponds to *width*, *height*, *depth* and *italic correction* parameters of individual glyphs. The *width* parameter corresponds to the *advanceWidth* value in OpenType's `hmtx` (horizontal metrics) table.

Height and *depth* information are not explicitly available in OpenType as there is some difference in the way TEX/Ω and OpenType deal with character dimensions. Figures 1–2 compare the two approaches, which we now describe further.

TEX and Ω have access to *abstract* height, depth and width parameters, in the sense that these values may be independent of the bounding box itself, of which they have no direct information. OpenType, on the other hand, provides bounding box, advance width and left and right sidebearing values so that both the width of the *abstract box* used for typesetting *and* the position of the glyph inside that box are known.

For OpenType fonts with TrueType outlines, the bounding box information is stored in the `glyf` table, while for CFF/OpenType, it needs to be extracted from the CFF data. Advance values and sidebearings come from either the `hmtx` or `vmtx` table, depending on the text direction. Nevertheless, the abstract height and depth parameters of TFM/OFM files are not present in OpenType in any form; vertical metric data such as advance height, top sidebearing, etc., are only used when typesetting vertically, they bear no relation to TEX's height and depth fields. Whether bounding box size parameters are acceptable as a substitute is also not evident. Nevertheless, this substitution has been used when adapting PostScript Type 1 fonts to TEX for many years already.

2. The JSTF table in OpenType contains prioritised suggestions where each suggestion defines the action that can be used to adjust a given line of text. When spaces are too large or too narrow, JSTF substitutions can help producing better lines. But this is by no means comparable to the global stretch and shrink parameters which give the maximum and minimum values of interword spaces.

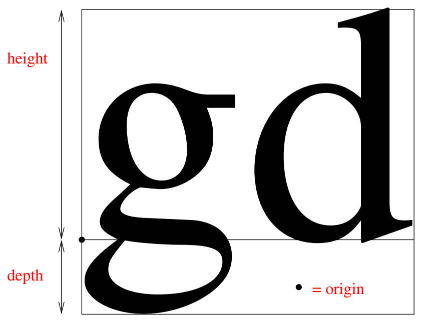

Fig. 1: TEX's notion of width, height and depth.

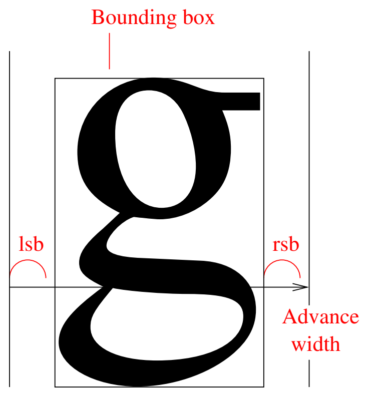

Fig. 2: OpenType's notion of *advance width* and *bounding box*.

Information regarding italic correction is not explicitly available in OpenType. However, it is possible to calculate it by subtracting the advance width value from *xMax*.

Information on glyph pairs refers to *ligatures* and *kerning*. Ligature substitution is handled by OpenType's `GSUB` table. It also supports contextual substitutions, which is similar to TEX's smart ligatures. Application of `GSUB` features will be discussed later in the article.

In OpenType, kerning information is available via both the `GPOS` and `kern` tables. In the former, a *Pair Positioning Subtable* specifies kerning data for glyph pairs, offering more flexibility than a typical kerning table: glyph pairs can be adjusted in both horizontal and vertical directions. It can also use the device table to adjust the positions of glyphs at each font and device resolution. The `kern` table, inherited from TrueType, provides a more classical approach to kerning; its use is discouraged.

OFM and OVF Creation

As explained above, OFM file creation for OpenType fonts is solely a temporary solution until direct OpenType access becomes available. Conversions described in the previous section are done by a Python utility called *makeovp*. The resulting OVP file can in turn be converted into OVF (virtual font) and OFM (virtual and real font metric) files by using the standard `ovp2ovf` utility.

In the virtual font, Unicode positions are mapped to glyph ID's according to the `cmap` table. Unencoded glyphs (swash alternates, contextual forms etc.) are automatically mapped to the private use area (PUA) of the Basic Multilingual Plane (U+E000...U+F8FF). The DVI file will use this "mixed" encoding scheme, by which *odvips* will later look up glyphs in the converted OpenType font (see section on PFC tables). It has to be stated again that this is only a temporary solution: our long-term plan is to eliminate the need for virtual fonts, as far as OpenType is concerned.

Advanced Typographic Table Support in Ω

This section presents how Ω makes use of OpenType's so-called *advanced typographic tables* such as GSUB and GPOS. As explained earlier, full support for these tables will certainly require a major rewrite of the Ω source code, which will most likely be a longer-term project. At the moment, only pair kerning information is used from the GPOS table (lookup type 2). If no GPOS table is present in the font, the `kern` table will be used instead, but only for TrueType outlines: CFF/OpenType fonts can only use the former.

The GSUB table is supported to a much greater extent: all kinds of lookups are handled with the exception of *Extension Substitution*. This is a major breakthrough that makes available such OpenType features as contextual ligature substitutions for Indic scripts, old style forms or small capitals in Latin, etc.

As a first step, OpenType substitution lookups need to be converted into ΩTP's (Ω Translation Processes, [3]) by a Python utility named *makeotp*. This program will extract every feature and lookup type of every script from the OpenType font file given as input and generate, for each one of them, a separate OTP file.[3] The naming convention for these ΩTP files is the following:

```
FontName-FeatName-LkupType-ScriptTag.otp
```

`FontName` is the name of the font. `FeatName` is the name of the feature that the ΩTP provides; feature names conform to Microsoft's feature tag registry. `LkupType` is the type of lookup the given feature can perform. An example ΩTP filename using the `dlig` (discretionary ligatures) feature for the *Palatino* (`pala.ttf`) font is `pala-dlig-lkup4-latn.otp`. Inside the newly generated ΩTP file, one finds expressions such as:

```
@"0051 @"0075 => @"E010;
@"0073 @"0070 => @"E026;
@"0073 @"0074 => @"FB06;
```

where multiple glyphs are being replaced by a single ligature glyph. The hexadecimal codes represent Unicode character positions, coming from the `cmap` table. Here, the third expression replaces 'st' with a single ligature glyph 'ﬆ' (mapped onto the PUA). Behold, Dear Reader, as this is the world premiere of an Ω document using OpenType fonts, with GSUB features applied!

`pala-frac-lkup5-latn.otp` is another example of a GSUB-originated ΩTP performing contextual substitution. This ΩTP contains the expressions of the form:

```
@"0034 @"002F @"0035 => @"2074 @"2044 @"2085;
@"0034 @"002F @"0036 => @"2074 @"2044 @"2086;
@"0034 @"002F @"0037 => @"2074 @"2044 @"2087;
```

where the first one replaces '4/5' by '⁴⁄₅' through a contextual substitution: `four`, `slash` and `five` are replaced by `foursuperior`, `fraction` and `fiveinferior` respectively.

Using GSUB *Features in Ω Documents*

While converting an OpenType font file, the *makeotp* utility will automatically generate ΩTP's for the GSUB features present in the font. In order to access these features, one must include a *style* file corresponding to the script and font. For a *(font, script)* pair, the `Fontname-LangTag.sty` style file is generated automatically by *makeotp*. For example, for the *CourierStd* (`CourierStd.otf`) font and *Latin*, the file created will be `CourierStd-latn.sty`. At this moment, the style file does not turn on GSUB features automatically, it is up to users to activate the features they need. One includes the style file in the Ω document header, i.e.,

```
\usepackage{CourierStd-latn.sty}
```

Then, to apply a specific feature, include

```
\pushocplist\CourierStddliglatn
```

at the point where the `dlig` (discretionary ligatures) feature of the Latin script is to be activated.

Devanagari with Ω: A Case Study

This section explains how to use ΩTP's for typesetting in the *Devanagari* script (especially in the Hindi language).

The following two input methods may be used:

- Velthuis Transliteration Scheme;
- a Unicode compliant editor.

3. In some cases, this operation will result in dozens of files per font. Do not forget: this temporary solution has been conceived to bring some key OpenType support to Ω as quickly and easily as possible.

The Velthuis scheme needs two ΩTP's to convert the input transcription into Unicode: `velthuis2-unicode.otp` and `hindi-uni2cuni.otp`, used to deal with *virama* and dependent vowels. In the case of the Hindi language, the final *virama* is removed by this ΩTP.

Things are easier using a Unicode (UTF-8) compliant editor: Ω can read Unicode-based text directly. In this case, the two additional ΩTP's used for the transliteration scheme are not needed. Only these two lines need be added to the header of the Ω document:

```
\DefaultInputMode UTF8
\InputMode currentfile UTF8
```

To turn on GSUB features inside Ω, the following line needs to be included in the header, supposing that we are using the `raghu.ttf` font:

```
\usepackage{raghu-deva.sty}
```

The *makeotp* program is intelligent enough to assemble this style file in such a way that OpenType features are applied in the correct order. For reference, the correct order for the Devanagari script is:

- `nukt`-Nukta form.
- `akhn`-Akhand Ligature.
- `rphf`-Reph form.[4]
- `blwf`-Below-base form.
- `half`-Half-form (pre-base form).
- `vatu`-Vattu variants.
- `pres`-Pre-base substitution.
- `abvs`-Above-base substitution.
- `psts`-Post-base substitution.
- `haln`-Halant form substitution.

The example below has been created using the *Raghu* OpenType font, with all GSUB features enabled. Note that, due to the lack of support for GPOS features, glyph positions may not always be correct.

भारत और रूस ने महत्त्वपूर्ण समझौते किये । भारत और रूस ने बुधवार को महत्त्वपूर्ण समझौतों पर हस्ताक्षर किये हैं जिनमें आतंकवाद से मुकाबला करने का एक संयुक्त समझौता भी शामिल है ।

Font and Glyph Outline Conversion

The following sections of the article deal with outline conversion issues. As it has already been pointed out, there are virtually no printers today with built-in OpenType font support. OpenType fonts will thus in all cases be converted before printing, either by a printer driver or by the application itself, to an appropriate format. In our case, no printer driver is available (native PostScript being produced), so we had to come up with our own font conversion procedure.

Theoretically, several destination formats are possible: PostScript Type 1, CFF (Type 2), Type 3, and even Type 42, for TrueType-flavoured OpenType. Each of these formats has its advantages and drawbacks:

- Type 1 enjoys complete support by all kinds of PostScript printers, including Level 1 printers. Its disadvantage is that conversion of TrueType-based OpenType fonts into PostScript is not trivial at all, and converting TrueType hints seems to be especially difficult;

- CFF is the deluxe edition of Type 1; it would be an ideal solution for Type 2 charstrings in CFF-based OpenType (no conversion is needed), but for TrueType outlines we face the same problems as with the Type 1 format. In addition, CFF needs Level 3 printers.

- although Type 3 is also universally supported, its major problem is that it is not suitable for PDF: Acrobat may render it very poorly. The PDF format has little support for Type 3 fonts, so that their glyphs end up being displayed as bitmap images. This not only affects rendering quality, but also obstructs glyph-to-character mapping mechanisms, so that Type 3 rendered glyphs cannot be searched, copied or indexed as characters;

- Type 42 could be a possible candidate for TrueType-flavoured OpenType: OpenType to TrueType conversion should not be difficult,[5] and the resulting TrueType font can easily be embedded in the Type 42 wrapper and then sent to the printer directly. However, a Level 2 printer is needed to interpret the Type 42 format, and furthermore, we know little about Type 42 compatibility with PostScript font operators such as `glyphshow` and `charpath`. Failure of these PostScript operators to work would mean that PostScript code using Type 42 fonts is of low quality.

After careful consideration of the advantages and disadvantages of each format, the two most promising choices seem to be Type 1 and Type 42. As our final decision, we opted for Type 1, mainly because of its simplicity and absolute support by printers — but this choice does not necessarily mean that Type 42 conversion will not be added later. If we consider discarding TrueType

4. The `rphf` and `blwf` features should not be used simultaneously. Two macros (in the form of ocplists) will be created, one with the `rphf` feature and the other with `blwf`, so that the user can switch between the two forms when necessary.

5. Speaking here of converting outlines and other basic data necessary for rendering, not about OpenType's advanced typographic tables.

hints acceptable (it should not matter for printed documents), OpenType to Type 1 conversion becomes a feasible, not too painful process.[6]

The solution we were looking for had to be intelligent enough to meet two important requirements: first, the same OpenType font should not be converted every single time it is used in a document, but *only once*, when it is first used. Secondly, to decrease the number of the resulting Type 1 fonts (and thus the size of the PostScript document), glyphs not referenced in the DVI file (not used in the document) should not be included in the Type 1 fonts. In other words, *odvips* should be able to do *subsetting*. These two points are justified by the following arguments:

- If the OpenType font is large (CJKV fonts can occupy tens of megabytes), converting can take an excessive amount of time;
- Type 1 fonts can encode up to 256 glyphs, while for OpenType this limit is 65,536. A CJKV OpenType font containing 10,000 glyphs would need to be converted into no less than forty Type 1 fonts.
- the download time of large fonts to the printer is slow, and printer memory is filled up quickly;
- we should keep the size of the final PostScript document as small as possible (Internet downloads, etc.);
- even if the OpenType file is small, why do the same conversion over and over again?

The method we propose is to convert OpenType fonts only at the time of their first use, and to store the converted Type 1 outlines in an intermediate format called *PFC* (abbreviation of “PostScript Font Container”, also a logical continuation of PFA and PFB). The PFC format can be regarded as a Type 1 glyph directory that provides one-by-one access to charstrings (as opposed to monolithic Type 1 fonts), allowing *odvips* to assemble Type 1 fonts on the fly, only including glyphs that are used in the given document (see fig. 3). If, say, a Chinese text contains only 500 different ideographs, *odvips* will retrieve the corresponding glyphs from the PFC font and create two Type 1 fonts (instead of forty!) to be included in the PostScript document. We will call these partial Type 1 fonts *mini-fonts*. The advantage of partial font creation is that both document sizes and printer overhead will be greatly reduced.

A Brief Description of the PFC format

We have chosen to design the PFC format to be compatible with the modular, table-based file structure of TrueType and OpenType (sometimes called *sfnt*)[7] so that all

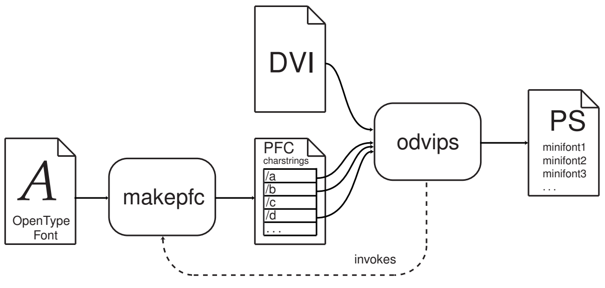

FIG. 3: Conversion and integration of OpenType font data into a PostScript document.

PFC data may be included, as tables, into the original OpenType font itself, instead of storing them in a separate file. However, if the font’s licence does not allow modification of the font file, we have no choice but to create a stand-alone PFC file.

Our PFC format defines the following tables:

- the header and table directory entries comply with OpenType’s header format, although the checksum is not used (at least for now);
- the `oMAP` table contains mapping information: for all glyphs in the PFC, their glyph code (the same code is used in the DVI file), the corresponding Unicode value, PostScript name, and the position of the glyph’s charstring length in the `oCHR` table;
- the `oCHR` table contains the length of charstrings as well as the charstrings themselves, already encrypted with *charstring encryption*;[8]
- `oGFD` (Global Font Data) includes general information needed in Type 1 minifonts, such as italic angle and bounding box values;
- `oPRI` and `oSUB` contain the beginning of the *Private dictionary* and the subroutines of the Type 1 minifonts, respectively. At the moment, these tables are unstructured and their contents, including the entire subroutine set, are embedded into each minifont without change. In the future, we plan to replace this dumb temporary solution with intelligent subroutine handling (partial subroutine download).

It is important to point out that the PFC charstring directory can be generated on demand, at the first use of a given OpenType font. If *odvips*, while reading the DVI file, finds a reference to an OpenType font, it will first check whether the corresponding PFC file already exists. If it does not, the OpenType-to-PFC conversion starts automatically.

Since the two steps of OpenType-to-PFC conversion and assembly of minifonts are more or less independent,

6. According to unofficial statements of Adobe people in the OpenType discussion forum, even Adobe’s “official” OpenType-ready applications like InDesign do an OpenType-to-Type1 conversion before printing.

7. See [1] for an overview of the OpenType file format.

8. See [2] for details.

there is no need to implement both inside *odvips*. In our solution, the converter is a separate utility called *makepfc* that is called by *odvips* when the font is used for the first time.

OpenType Outline Conversion

Our work[9] related to the *makepfc* utility revolves around the conversion process of TrueType quadratic splines to cubic Bézier curves of the PostScript Type 1 format. We have decided to build our work on Just van Rossum's *ttx* utility as the parser, mainly because of its ability to read OpenType tables, as well as its easy extendability to new functionalities. The conversion itself, as far as TrueType outlines are concerned, is based on the *ttf2pt1* program.

The conversion algorithm for TrueType fonts goes as follows: suppose we are given a TrueType (i.e., quadratic) curve whose starting and ending points are `startx` and `endx` respectively, with `ctrlx` as the single control point. To split the single control point into two, as needed for the Bézier cubic, the following calculations need to be done (fig. 4):

- starting and ending points, i.e., `startx` and `endx`, remain the same;
- the two control points are given by $(2 * \texttt{ctrlx} + \texttt{startx})/3$ and $(2 * \texttt{ctrlx} + \texttt{endx})/3$.

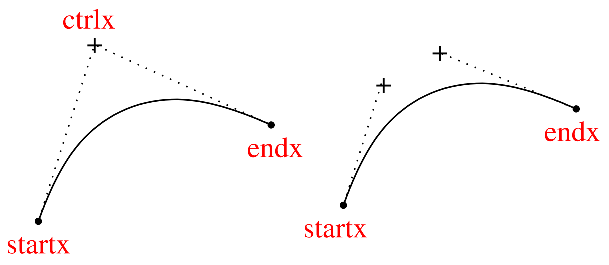

Fig. 4: Quadratic TrueType and cubic Bézier curves.

For CFF/OpenType fonts, no outline conversion is needed, except for Type 2-specific charstrings like `flex` or `random`, where conversion can be difficult: the `random` operator, for example, has no Type 1 equivalent at all. Most information, however, can be directly copied from OpenType's CFF table.

Configuration of odvips

To understand exactly how *odvips* deals with OpenType fonts, let us suppose that we are using `verdana.ttf` as our OpenType font, and that a DVI file using Verdana's original OpenType glyph ID's has already been created by Ω (either by means of a virtual font or by accessing OpenType tables directly). *odvips* will thus find a `verdana` font referenced inside the DVI, where `rverdana` is the TeX name (the name of the corresponding OFM file) of the OpenType font. It will therefore look through `psfonts.map`, trying to find an entry with the same fontname. The following line would be added to `psfonts.map`:

```
rverdana verdana <verdana.ttf
```

odvips will then realise that `verdana` is a TrueType-flavoured OpenType font and that it needs to find its PFC charstrings to be able to continue. First, it will try to find the PFC tables inside `verdana.ttf`. If there are no PFC tables in `verdana.ttf` (which is very probable as—due to Microsoft's restrictive licence—we are not allowed to modify the font) or if *odvips* cannot even find the file, it will look for a separate file called `verdana.pfc`, supposing that it must have already been created. If the PFC is not found either, *odvips* will invoke the *makepfc* program to produce the PFC tables on the fly. If both files are missing, *odvips* falls back to a default font.

A Final Remark

The process described above (including metric, style and OTP file creation, font conversion, modification of configuration files, etc.) may seem complicated. And, to be honest, it is, even if the individual steps are easy to carry out. We therefore plan to write a single script that would call the individual *make** and other programs automatically and thus let the user do the whole conversion process in a single step.

References

[1] The OpenType Specification v1.4. `http://www.microsoft.com/typography/otspec/default.htm`

[2] Adobe Type 1 Font Format. `http://partners.adobe.com/asn/developer/pdfs/tn/T1Format.pdf`

[3] Draft Documentation for the Ω system. 7 March 1998. `http://www.loria.fr/services/tex/moteurs/omega7mar1998.pdf`

[4] *Haralambous, Yannis, Plaice, John*: Low-level Devanagari Support for Omega—Adapting devnag. *TUGboat* 23(1), 2002, Proceedings of the TUG Annual Meeting, pp. 50–56. `http://omega.enstb.org/yannis/pdf/tug2002.pdf`

9. Anish Mehta (TTF) and Ghislain Putois (CFF) are the two developers of the *makepfc* utility.