Ω/CHISE: A Typesetting Framework based on the Character Information Service Environment

Izumi Miyazaki and Toru Tomabechi

Abstract

The open-set scheme proposed by CHISE can (at least theoretically) handle an infinite number of characters and should come to a rescue for those who have felt the inconvenience of the CCS-based text-processing. As a part of the CHISE Project and as a sample application of this environment to the real-world computing, the present authors have been undertaking the development of the Ω/CHISE system which combines Ω's typesetting capability and CHISE's power in basic text-processing. This paper will describe how Ω/CHISE's works and how it is implemented.

1 Introduction

The CHISE (CHaracter Information Service Environment) project initiated by Tomohiko Morioka aims at a radical reconstruction of text-processing environment. Treating a character in terms of "feature bandle" rather than as a specific code point (and its associated glyph) in a coded character set (CCS), CHISE provides a powerful framework that overcomes the limitation imposed by the traditional model ([1]. See also the article by S. Moro in this volume). Though the discussion within the CHISE circle often tends to get rather abstract (or "philosophical," whatever the term may mean), the present paper will advisedly leave out such a subject—interesting as it might be—and will try to be as down-to-earth as possible. It's all about how we obtain *one* of the possible practical results of CHISE text-processing, namely the typeset/printed output: nothing more, nothing less.

* * *

Just as in any other area of text-processing, existing typesetting systems are fundametally bound up with the traditional CCS scheme (including Unicode/ISO-10646) and are designed to handle characters only within an artificially imposed limitation. The problem of this traditional model become especially apparent when processing texts written in CJK languages where users quite often encounter with the characters that cannot be properly encoded due to their absence in the existing character sets and have to have recourse to some idiosyncratic solutions, such as the so-called *Gaiji*. Even today, the situation cannot be said to be very different from that of the bad old days of *Gaiji* proliferation. As Unicode's character inventory continues to grow larger, many people come to hope that one day we will be able to cover all known characters. This may be partially true, but will not solve the intrinsic problem of the CCS model.

The open-set scheme proposed by CHISE can (at least theoretically) handle an infinite number of characters and should come to a rescue for those who have felt the inconvenience of the CCS-based text-processing. As a part of the CHISE Project and as a sample application of this environment to the real-world computing, the present authors have been undertaking the development of the Ω/CHISE system which combines Ω's typesetting capability and CHISE's power in basic text-processing. In the following, this paper will describe how Ω/CHISE works and how it is implemented.

2 Why Ω?

Ω is an extension to Donald E. Knuth's TeX typesetting system which is being developed by John Plaice and Yannis Haralambous.[1] While inheriting the precision and the flexible typesetting capability of TeX, Ω realizes additional capabilities of text-processing, which make this typesetting engine quite suitable for our purpose. To our project, the following features are especially relevant:

[1] `http://omega.cse.unsw.edu.au:8080/index.html` (As of writing of this paper, the web site seems to be down.) Ω is a software still under development and, unfortunately but inevitably, there is no definite documentation of its functionality.

Large code space: Original TEX can handle only 256 code points per font and local CJK extentions, such as pTEX, typically extend the code space to 94×94 grid (=8836 cells). The current implementation of Ω has full 16bit code space[2] and thus capable to handle 65536 characters in a single font.

External ΩTP: ΩTP stands for "Ω translation process," which is of two types, viz. "internal" and "external." The both types of ΩTPs are used for pre- or post-processing of input/output stream to/from the so-called "stomach" of Ω typesetting engine. While the internal ΩTP is implemented as a special built-in programming language in Ω, the external ΩTP can be any type of STDIN-STDOUT filter written in C, Perl, Ruby, Python, or whatever. The external ΩTP facility allows the user to extend the capability Ω typesetting engine and makes it suitable for complex tasks.

As we are dealing with an open-set system, the absolute number of characters per font is *not* really essential, however large it may be. However, from the practical point of view, it is much easier to have a small number of large fonts than to have to deal with a huge bunch of small ones.[3] The 16bit code space of Ω meets this requirement at least at the current time.

In order to draw forth the potentiality of CHISE in the domain of typesetting, the external ΩTP plays important roles. To make use of character information which characterize CHISE, a flexible interface between CHISE database and the typesetting engine is necessary. Furthermore, Ω/CHISE's most important feature— on-the-fly generation of missing glyphs using KAGE—involoves a network transaction. Thanks to the external ΩTP facility, these two features can be easily implemented.

And finally let us add this: we shall not forget that the CHISE project is an Open Source project. Ω, as its predecessor TEX, is Open Source: for our purpose, it is the best available typesetting engine.

3 How it works

Now, let us explain how Ω/CHISE works. First, we explain the outline of typesetting process in this section, and the detail of each components will be treated in the next.

As shown in Figure 1, Ω/CHISE system consists of several components. The core of the system is a macro package named `chise.sty` from which two external ΩTP are called.

The entire process is roughly divided into three phases, 1. typesetting, 2. glyph generation and 3. DVI to PDF conversion, as described below:

Typesetting While reading in the source file, the first ΩTP is called and scans CJK characters and IDS (Ideographic Description Sequence). Each time a CJK character or IDS is encountered with, the ΩTP sends query to the character database relevant in the current font environment in order to see whether the character or IDS in question has a corresponding glyph in the target font. The next behavior of the ΩTP is determined according to the query result, which can be classified as follows:

1. Source: CJK character
   (a) Query result: YES
   (b) Query result: NO
2. Source: IDS
   (a) Query result: YES
   (b) Query result: NO

In case 1 (a), ΩTP leaves the input character as is. In case 2 (a), the input IDS is replaced with the corresponding character. In these two cases, the character is typeset using the glyph in the target font which is determined by the current font environment.

In case 1 (b), ΩTP asks the database for the IDS corresponding with the character and replaces the character in the source file with a macro format which specifies the font and code position to be used for typesetting the character. In case 2 (b), the IDS is replaced with the macro code. In the both cases,

[2] It is planned that the code space will be expanded to 32bit in a future version. Though Ω is often described as using Unicode internally, this is rather misleading. Ω indeed has a 16bit code space which allows to use BMP in a single font file, but it does not follow from this that Ω is bound up with some specific CCS such as Unicode.

[3] As we will explain below, the actual implementation of Ω/CHISE *does* use a rather large number of sub-fonts. However, those sub-fonts are bundled together in larger fonts via virtual font facility and hidden from the ordinary user.

the IDS in question is written out in a auxiliary file which will be used later for on-the-fly glyph generation.

Let us review the process described above in a more concrete manner with the example shown in Figure 2:

- The first 知 character, which is in the grobal font environment (specified as package option to `chise.sty`) of the source file, is typeset using the glyph in a GB font. On the other hand, the second 知 character is enclosed within a local font environment (JIS in this example) and typeset using the glyph in a JIS font. [Case 1 (a)]
- Then, the IDS consisting of IDC U-2ff0, 矢 and 口 characters is replaced with a single character 知 and typeset using the glyph in JIS font as in the previous example. [Case 2 (a)]
- However, the next IDS, U-2ff1, U-2ff0, 口, 口,U-2ff0, 口 and 口, (usually) has no corresponding character in the target font (JIS). In this case, the IDS is written out to an auxiliary file and replaced in the input stream with a macro command. The character is typeset using the glyph generated on-the-fly. [Case 2 (b)]
- And the last character, which is in a simplified Chinese form but enclosed in JIS environment, cannot be found in the target font. In this case, the ΩTP tries to get the corresponding IDS from the database and writes it out to the auxiliary file. The character is typeset in the same manner as in the previous case. [Case 1 (b)]

Apart from the character/IDS handling, the typesetting process of the document goes normally just in the same way as with the ordinary Ω/TEX and generates a DVI file.

Glyph generation Once the typesetting process is finished, the second ΩTP is called as an `\end{document}` hook. The ΩTP reads in the auxiliary file written out during the typesetting process. According to the IDS information of the missing glyphs, the ΩTP sends glyph requests to KAGE ([2]. See also the article by K. Kamichi in this volume) over the http communication. Then the ΩTP converts glyph outlines in SVG format returned by KAGE into PostScript Type 1 human-readable codes and, with the help of two external programs (t1asm and pfaedit) creates Type 1 font(s) in PFB format.

DVI to PDF conversion Finally, the created fonts are used in the process of DVI→PDF conversion by dvipdfmx,[4] and included in the resulting PDF.

4 Implementation

Now let us see how each component is implemented.

4.1 Macro package: `chise.sty`

This is the core component of Ω/CHISE system. It consists of a series of macro definitions that controls the behavior of the two external ΩTPs which will be explained later.

`chise.sty` accepts the following package options:

- File coding system:
  This is a required option used for specifying the coding system of input file. Currently, the macro package accepts four UTF-8 variants used in XEmacs/CHISE.
  - `utf8mcs`
  - `utf8cns`
  - `utf8gb`
  - `utf8ks`
- Global font environment:
  This option is used for specifying the document's default font environment. Option's names must self-explanatory.
  - `gbfont`
  - `cnsfont`
  - `jisfont`
  - `ksfont`

[4] A CJK extention to Mark A. Wicks' dvipdfm, which is capable of handling DVI files generated by Ω. See `http://project.ktug.or.kr/dvipdfmx/`

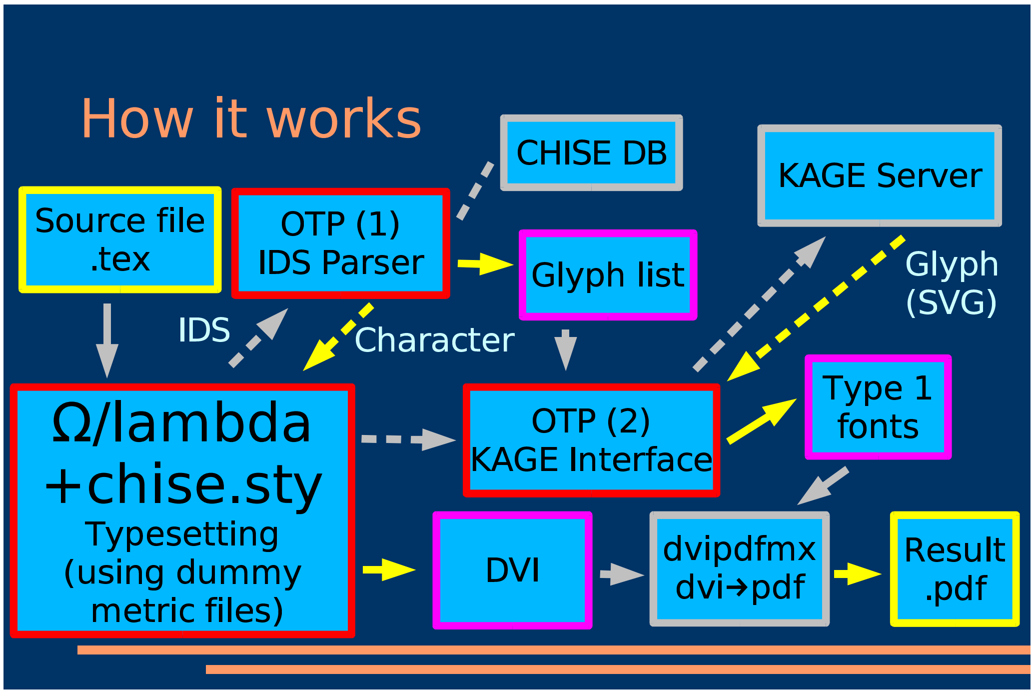

Figure 1: How Ω/CHISE works

```
\documentclass{article}
\usepackage[utf8mcs,gbfont,kage]{chise}
\begin{document}
知
\begin{JISfont}
知 ⿰矢口 ⿱⿰口口⿰口口 纩 ...
\end{JISfont}
\end{document}
```

Figure 2: Sample input file

- KAGE use:
  If this option is given, Ω/CHISE interacts with KAGE server for glyph generation. Omit this if you are processing a draft document and do not want to generate glyphs.
  - `kage`

And `chise.sty` defines the following macro commands available to users:

- Font changing commands:
  Locally changes the CJK font environment. Self-explanatory.
  - `\jisfont{}`
  - `\gbfont{}`
  - `\cnsfont{}`
  - `\ksfont{}`
  - `\multifont{}` — Try to use all available CJK fonts whatever the global font environment may be.
- Entity reference:
  - `\ER{}`
    For example, `\ER{U-4e00}` stands for Unicode/ISO-10646's code point 4e00 and prints the character "一". This command can also be used for expressing IDC, as in `\ER{U-2ff0}矢口`
- IDC printing commands:
  Prints IDC verbatim.
  - `\idcltr`
    U-2ff0
    IDEOGRAPHIC DECSRIPTION CHARACTER LEFT TO RIGHT
  - `\idcatb`
    U-2ff1
    IDEOGRAPHIC DECSRIPTION CHARACTER ABOVE TO BELOW
  - `\idcltmr`
    U-2ff2
    IDEOGRAPHIC DECSRIPTION CHARACTER LEFT TO MIDDLE AND RIGHT
  - `\idcatmb`
    U-2ff3
    IDEOGRAPHIC DECSRIPTION CHARACTER ABOVE TO MIDDLE AND BELOW
  - `\idcfs`
    U-2ff4
    IDEOGRAPHIC DECSRIPTION CHARACTER FULL SURROUND
  - `\idcsfa`
    U-2ff5
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM ABOVE
  - `\idcsfb`
    U-2ff6
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM BELOW
  - `\idcsfl`
    U-2ff7
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM LEFT
  - `\idcsful`
    U-2ff8
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM UPPER LEFT
  - `\idcsfur`
    U-2ff9
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM UPPER RIGHT
  - `\idcsfll`
    U-2ffa
    IDEOGRAPHIC DECSRIPTION CHARACTER SURROUND FROM LOWER LEFT
  - `\idcol`
    U-2ffb
    IDEOGRAPHIC DECSRIPTION CHARACTER OVERLAID

In addition to the font changing macros described above, the font environment can also be changed locally by `\begin{}...\end{}` style command:

- `\begin{JISfont}`
  ...
  `\end{JISfont}`
- `\begin{GBfont}`
  ...
  `\end{GBfont}`
- `\begin{CNSfont}`
  ...
  `\end{CNSfont}`
- `\begin{KSfont}`
  ...
  `\end{KSfont}`
- `\begin{Multifont}`
  ...
  `\end{Multifont}`

4.2 ΩTP (1)—CJK character/IDS parser

The first ΩTP is a Perl script, named "inCHISE", which scans and transforms the input stream. As has already been explained, the main task of this script is to parse CJK characters and IDS and to query databases.

This ΩTP has to change its behavior according to the coding system of the input file and the font environment in which the current part of document is being processed. With stand-alone scripts, such behavior changes can be easily effectuated by simply passing relevant options to the script. However, the current implementation of Ω does not allow passing options from within the style file to the external ΩTPs and we are forced to adopt a less than smart way of work-around by creating symbolic links from the script to the following file names:

- Utf8cnsToUniCNS
- Utf8cnsToUniGB
- Utf8cnsToUniJIS
- Utf8cnsToUniKS
- Utf8cnsToUniMulti
- Utf8gbToUniCNS
- Utf8gbToUniGB
- Utf8gbToUniJIS
- Utf8gbToUniKS
- Utf8gbToUniMulti
- Utf8jisToUniCNS
- Utf8jisToUniGB
- Utf8jisToUniJIS
- Utf8jisToUniKS
- Utf8jisToUniMulti
- Utf8ksToUniCNS
- Utf8ksToUniGB
- Utf8ksToUniJIS
- Utf8ksToUniKS
- Utf8ksToUniMulti
- Utf8mcsToUniCNS
- Utf8mcsToUniGB
- Utf8mcsToUniJIS
- Utf8mcsToUniKS
- Utf8mcsToUniMulti

The first part of each file name represents the coding system of the input file and the last part the target font (for example, Utf8cnsToJIS: input file is encoded in utf8-cns—specified in package option to `chise.sty`; use JIS font to typeset characters—specified as global or local font environment in the source file). In the actual process, the script is called by one of these names: that name is passed to the script via `$argv[0]` which is then used to determine what the script's behavior should be.

In the example in Figure 2, the default file name of the ΩTP is Utf8mcsToGB as the two package options `utf8mcs` and `gbfont` are given. When the first 知 character, which is to be typeset using a GB font, is processed, this ΩTP queries the database created from Adobe CMAP for GB character set in order to see whether 知 character is available in the GB font installed in the system. Then, when the second 知 enclosed in JIS font environment is processed, the ΩTP is called under the name Utf8mcsToJIS. The ΩTP now queries another database which is based on CMAP for JIS.

4.3 ΩTP (2)—KAGE interface

The second ΩTP, which is called from `chise.sty` at the end of typesetting process as an `\end{document}` hook, has two tasks. The first of the two is to interact with KAGE server over HTTP communication, the second to generate PostScript Type 1 fonts.

When called, the ΩTP tries to read in the auxiliary file created by the first ΩTP during the typesetting session. The auxiliary file contains a list of IDSs for characters which are absent from the exisiting character sets and, therefore, cannot be typeset using pre-existing fonts. The ΩTP first converts the raw IDS encoded in UTF-8 into the URI form acceptable for KAGE. For example, the second IDS in Figure 2, U-2ff1 U-2ff0 □□ U-2ff0 □□, is converted into

`u2ff1u2ff0u53e3u53e3u2ff0u53e3u53e3`.[5]

Then, appended to HTTP prefix, the above query string is sent to the server on which KAGE is running.

The second task of this ΩTP is to create PostScript Type 1 fonts. If KAGE successfully creates and returns the requested glyph, the returned data, which is a glyph outline in SVG format, is converted into Type 1 charstring and stored in an array. The glyph outline is also cached in an external database on the local system in order for the process to be speeded up when the same glyph is requested next time.

Once all the necessary glyphs are obtained, the ΩTP bandles those glyphs into Type 1 human-readable fonts. These fonts are passed through an external helper program, t1asm,[6] to be compiled into PFB format. Further, the resulting PFB fonts are "cleaned up" by pfaedit[7] which is run in the script mode. This latter process eliminates redundant control points and adds hints to the glyphs.

CJK typsetting usually requires two kinds of glyph designs, namely, Mincho/Myongjo/Song face and Gothic/Heiti face. However, the current implementation of Ω/CHISE does not store the typeface design information during the typesetting process—we simply create the both for each character at the same time.

4.4 Fonts

As explained above, we use PostScript Type 1 format for the fonts created by the second ΩTP. A Type 1 font can contain only 256 glyphs which are available at one time for typesetting.[8] However, typesetting of a CJK document can possibly require more than 256 glyphs thus created. We therefore bandle multiple Type 1 fonts together into a large virtual font, which, in the current implementation of Ω, can contain at most 256 sub-fonts or 65536 glyphs. We define two virtual fonts for each typeface design (Mincho and Gothic), i.e. four virtual fonts in total. This allows $65536\times2=131072$ glyphs to be used for each typeface. This should be reasonably sufficient for normal use, and, if need be, a simple modification to the second ΩTP allows more glyphs to be used.

The four virtual fonts are named respectively chise000min.ovf, chise001min.ovf, chise000got.ovf and chise001got.ovf. Under each virtual font are included sub-fonts named chisesub000min ... chisesub255min, chisesub256min ... chisesub511min, etc. (See Figure 3)

We prepare beforehand the dummies for all the $256\times2\times2=1024$ sub-fonts that are placed in system's TEXMF tree. This is necessary because at the moment the document is being typeset, we cannot know how many glyphs will be needed. When the typesetting process is terminated and the necessary fonts are created, the DVI-PDF conversion program (dvipdfmx) first looks at the current working directory and uses the newly created fonts which are to be found there: they override the dummy fonts.

5 Concluding remarks

In guise of conclusion, we shall enumerate major problems and TODOs of Ω/CHISE implementation.

[5] This form of query string is compatible with KAGE 0.3 and now became obsolete. The implementation of the ΩTP will shortly be adapted to a new version of KAGE.

[6] Included in t1utils package.
See `http://www.lcdf.org/~eddietwo/type/index.html#t1utils`

[7] See `http://pfaedit.sourceforge.net/`

[8] More precisely, more than 256 glyphs can in fact be *defined* in one Type 1 font, but only 256 at most of them can be *accessed* through the encoding vector.

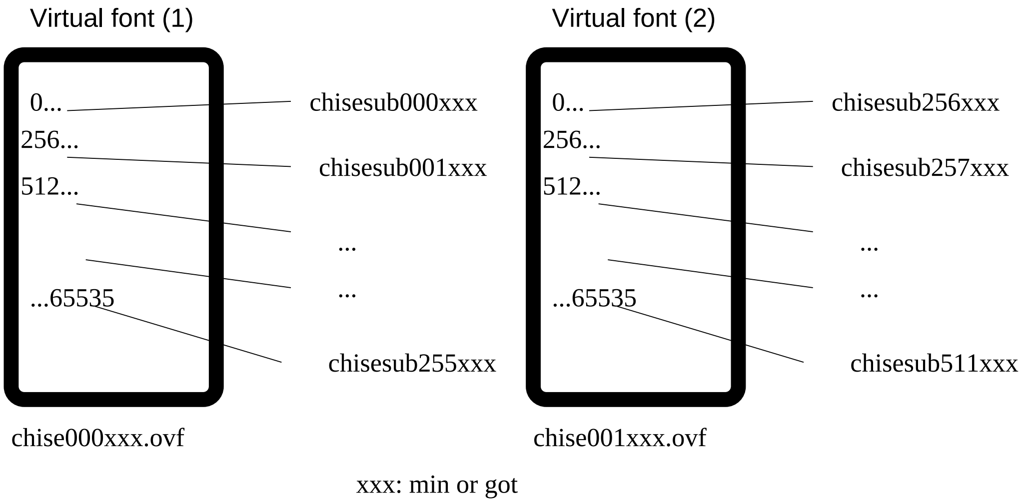

Figure 3: Virtual font composition

First, we should point out problems pertaining to the current implementation of Ω itself. It is preferable that the external ΩTPs be placed in the system's TEXMF tree (e.g. in `/usr/local/share/texmf/omega/...`) and Ω itself can find them without specifying where they are, just as with internal ΩTPs/ΩCPs. Currently, however, it is necessary to provide the full-path to the external ΩTPs in the style-file. Furthermore, as we explained above, it is currently impossible to pass options/parameters to the external ΩTPs from within the style-file. We can only hope these two problems will be solved in a future version of Ω.

As for the font generation, there are two points to be improved. First of all, we should catch up the development of KAGE and be able to use fully its functionality. The current implement of the Ω/CHISE does not utilize KAGE's glyph design parameter, which allows a fine tuning of glyph design preference which is varied among the CJK languages. Secondly, it is necessary to improve the quality of generated glyphs. This is largely the matter of KAGE development, but Ω/CHISE side may have to do at least one thing. The glyph outlines that KAGE returns have countour-overlaps which appear as blanks when printed. We tried the overlap-removal function of pfaedit, but it does not function properly, at least in the current version. It is therefore necessary to find some work-around. However, given that the overlap-removal algorithm is a rather complicated one and quite hard to write such a routine that works correctly, we find it not a good idea to create our own (and we are not much interested in that anyway...). We do not know what to do for the moment: maybe we had better wait calmly until pfaedit improves its overlap-removal.

Ω/CHISE is still in an experimental stage. Though it works somehow, the current status is far from being satisfactory. We shall continue the work so that the system can meet the needs of potential users (if any).[9]

Bibliography

[1] T. Morioka, *et al.*, "CHISE Project", in *Journal of Japan Association for East Asian Text Processing* 4 (2003), 58–69.

[2] K. Kamichi, 「漢字フォント自動生成サーバ "KAGE" の構築— 文字コードの枠組みを超える次世代漢字処理の提案—」, in *Journal of Japan Association for East Asian Text Processing* 3 (2002),4–13.

[9] Ω/CHISE is available at: `http://cvs.m17n.org/cgi-bin/viewcvs/omega/?cvsroot=chise`