A Multidimensional Approach to Typesetting

John Plaice, Paul Swoboda
UNSW, Sydney, Australia

Yannis Haralambous
ENST Bretagne, Brest, France

Chris Rowley
Open University, London, UK

Origins of the Research

- Discussions between Ω Project and LaTeX3 Project.
- Initiated at *Int. Symp. on Multilingual Inf. Proc.*, Tokyo, January 2001, organized by NAIST (formerly ETL).
- Attempt to find a low-level Ω interface that could be used for higher-level multilingual LaTeX programming.
- Ω waiting for LaTeX API in order to implement ...
- LaTeX waiting for working Ω in order to experiment ...
- We needed a model for change of language and script supporting complex variants.

The Answer: Intensional Programming

- Programs adapt to multidimensional *context*.
- Dimensions act as distributed global variables.
- The context *permeates* everything, and each component adapts accordingly by testing the context.
- Components come in multiple *versions*, i.e., (*context*, *object*) pairs; the *most relevant* is chosen as needed.
- A change of value for a dimension can simultaneously affect multiple levels of thousands of programs, with the support of a networked context server.

But *What* is the problem?

The Standard Model

- Unicode: one character set for all of the world's characters.
- XML: to hold tree-structured documents.
- ISO-639: two-letter codes for the world's languages.
- XSL-FO: to do the formatting.

The Standard Model

- Unicode: one character set for all of the world's characters.
- XML: to hold tree-structured documents.
- ISO-639: two-letter codes for the world's languages.
- XSL-FO: to do the formatting.

In this model, it is "understood" that character, glyph, language and script are discrete, eternal, unchanging entities.

History disappears, as does cross-cultural evolution.

Language Evolution

- English is a multidimensional complex.
- *Time*: Old, Middle and Modern English.
- *Space*: national and regional Englishes.
- *Culture*: science, arts, business, diplomacy.

Implication: spellings and hyphenations vary, hence different rules must be used according to the situation.

Script Evolution

- There are some 200 named Indic scripts.
- Each has evolved from the original Brahmi script.
- All have a similar, but not identical, structure.
- Unicode encodes only a dozen of these scripts.

Implication: Each of these scripts must be understood both as a separate script as well as a variant of a single script.

Language/Script Co-Evolution

- Turkish: Arabic ⇒ Latin script, with extra letters.
- German: Gothic ⇒ Latin script, without short/long s.
- Chinese: traditional ⇒ simplified characters.
- Chinese Latin transliteration: Wade-Giles ⇒ *pinyin*.
- Berber: Tifinagh (not in Unicode!), Arabic and Latin.

Implication: Need to allow semi-automatic conversion from one writing system to another, using morphological analyzers.

Character/Glyph Co-Evolution

- A character is a unit of "textual" information.
- A glyph is a unit of "visual" information.
- The unit "æ" is a Danish character, but an English ligature.
- Chinese variant characters can be seen either as glyphs or characters, depending on the situation.
- When typesetting continuous scripts, the glyphs do not necessarily even correspond to characters, but, rather, to bits thereof.

Implication: glyph and character are fluid concepts.

How do we deal with this variance?

Our Initial Model

- We use a tree-structured context:

```
<characterset:<Unicode + encoding:<UTF8>> +
 input:<XML + DTD:<TEI>> +
 language:<English +
           spelling:<Australian> +
           script:<Latin>> +
 output:<PDF +
         viewer:<AcrobatReader +
                 version<5.0 + OS:<MacOSX>>>>>
```

- `input` and `output` are *dimensions*.
- `lang:script` is a *nested dimension*.

Context-Dependent Processing (1)

An initial context is set up from:

- environment variables
- system locale
- user profiles
- command-line arguments
- menu selections
- document markup

Context-Dependent Processing (2)

The context is active throughout the process. It is used:

- to interpret the input;
- to choose the exact format for the output;
- to determine what processing should be undertaken.

Defining the process involves:

- choosing the number of passes;
- choosing linguistic, layout, or other plug-in tools;
- parametrizing each of these.

Context-Dependent Processing (3)

Changing language, script or font means changing the context.

- Context operations *modify* contexts:

```
[language:<French +
          spelling:<Quebec> +
          script:<Latin>]
```

- Flexible approach that can be used to specify subtleties of the typesetting process, to an arbitrary level of precision.
- Specialized tools can be invoked as needed.

Adding Contexts to Ω (1)

- `\contextset{`C_{op}`}` modifies the current context.
- `\contextbase{`D`}` gives the value for dimension D.
- `\contextshow{`D`}` gives the sub-context for dimension D.

where C_{op} is a context operation and D is a nested dimension.

Adding Contexts to Ω (2)

Three mechanisms to adapt Ω's behavior:

- versioned execution flow;
- versioned macros;
- versioned ΩTPs.

Versioned Execution Flow

If C is the current context, then executing:

$$\texttt{\textbackslash contextchoice\{\{}C_{\mathrm{op}_1}\texttt{\}=>\{}exp_1\texttt{\},}$$
$$\ldots$$
$$\texttt{\{}C_{\mathrm{op}_n}\texttt{\}=>\{}exp_n\texttt{\}}$$
$$\texttt{\}}$$

will select and expand one of the expressions exp_i. The one chosen will correspond to the *best-fit* context among $\{C\ C_{\mathrm{op}_1},\ \ldots,\ C\ C_{\mathrm{op}_n}\}$

Versioned Macros (1)

If C is the current context, then:

$$\texttt{\textbackslash vdef\{}C_{\mathrm{op}}\texttt{\}\textbackslash}\mathit{controlsequence}\ \ \mathit{args}\texttt{\{}\mathit{definition}\texttt{\}}$$

defines the $C\ C_{\mathrm{op}}$ version of *controlsequence*. Definitions are scoped as for TEX.

Versioned Macros (2)

The standard TEX definition:

`\def\`*controlsequence* *args*`{`*definition*`}`

is simply equivalent to

`\vdef{<>}\`*controlsequence* *args*`{`*definition*`}`

i.e., it defines the empty version of a control sequence.

Versioned Macros (3)

If *controlsequence* is defined for contexts $\{C_1, \ldots, C_n\}$,
and C is the current context, then

$$\backslash \mathit{controlsequence}$$

will select the definition corresponding to the *best-fit* context:

$$\max\{C_i : C_i \sqsubseteq C\}$$

A particular version can also be requested:

$$\texttt{\textbackslash vexp\{}C_{\mathrm{op}}\texttt{\}}\backslash \mathit{controlsequence}$$

Versioned ΩTPs

An ΩTP is a filter, reading from standard input and writing to standard output. An ΩTP-list is a series of ΩTPs.

- Internal ΩTP — Rules can be preceded by a context:

  `<<`*context*`>>` *pattern* `=>` *expression*

  A rule is only examined if its context is less refined than the current context.

- External ΩTP — Context is passed as an argument:

  program `-context=`*context*

The User-Level Interface

Ω always has an active ΩTP-list, and the `\contextchoice` operator can be used to build ΩTP-lists that adapt to the context, by selectively turning on and off the member ΩTPs.

Finally, Ω has a user-level means for manipulating the large sets of parameters that must be handled when doing complex multilingual typesetting: parameters are changed as needed.

Versioning of the macros and ΩTPs allows one to deal with the variance in language and script, as well as encouraging the sharing of resources across multiple languages.

We now have a

context-dependent *user interface.*

How about a
context-dependent *typesetter*?

Multiple-Phase Character-Level Typesetter

We present a simple approach:

- *Preparation*: Adding extra markup to input stream.
- *Segmentation*: Cutting input into typesettable *clusters*.
- *Cluster typesetting*: Finding formattings for clusters.
- *Recombination*: Relative positioning of formatted clusters.

Example typesetter

```
stream<Glyph>
typeset(stream<Char> input, Context context) {
    stream<Char> prepared =
        input.apply(otp_list.best(context));
    stream<Cluster> segmented =
        segmenter.best(context)(prepared);
    stream<TypesetCluster> typeset =
        clusterset.best(context)(segmented);
    stream<Glyph> recombined =
        recombine.best(context)(typeset);
    return recombined;
}
```

Preparation

```
stream<Char> prepared =
    input.apply(otp_list.best(context));
```

The preparation phase works entirely on *characters*, i.e., at the *information exchange* level, but it allows additional typographic information to be added to the character stream, so that the following phases can use the extra information to produce better typography.

Segmentation

```
stream<Cluster> segmented =
    segmenter.best(context)(prepared);
```

The segmentation phase splits the stream of characters into clusters of characters; typically, segmentation is used for word detection. Trivial process for English, non-trivial for languages such as Thai.

This process can also be used to find compound word divisions in Germanic and Slavic languages.

Choice of segmenter is clearly context-dependent.

Cluster Typesetting

```
stream<TypesetCluster> typeset =
    clusterset.best(context)(segmented);
```

A *cluster engine* processes a character cluster, taking into account the current context and produces the typeset output — a sequence of positioned glyphs.

When hyphenation or some other form of cluster-breaking is allowed, there are multiple possible typeset results, and all of these possibilities must be output.

For complex scripts and dynamic fonts, many different engines are needed.

Recombination

```
stream<Glyph> recombined =
    recombine.best(context)(typeset);
```

Typeset clusters are placed next to each other. For simple text, such as English, this simply means placing a fixed stretchable space between typeset words. In situations such as Thai and some styles of Arabic typesetting, kerning would take place between words. Once again, the recombiner's behavior is context-dependent.

But don't `Char` and `Glyph` vary with the context?

Problem with Example

```
stream<Glyph>
typeset(stream<Char> input, Context context);
```

In this type declaration, the types `Glyph` and `Char`, are fixed, unchanging sets, not at all consistent with the view that character and glyph are multidimensional entities.

However, if these basic types were to continually change, then no algorithms could be written, because one could never be sure of the "atoms" with which one was working.

Typographical Spaces

The typographical space constrains the variance in the context.

```
stream< Glyph<TS> >
typeset(stream< Char<TS> > input, Context context);
```

In a typographical space, certain parameters are kept fixed, or at least their values are kept within a certain range. Other parameters may vary at will, and their values may be manipulated by the algorithms within that space.

Typesetting Unicode

The typographical space is a necessary solution to the problem raised by the existence of multi-script character sets such as Unicode. It is simply infeasible to write a single typesetter that will do quality typesetting of Egyptian hieroglyphics, Japanese kanji with furigana, Persian in Nastaliq style, and German using Fraktur fonts.

By creating separate typographical spaces for these different kinds of situation, we can allow specialists to build typesetters for the scripts and languages that they know best.

How to build a multilingual typesetter

What is still needed for quality multilingual typesetting is to define some basic parameters, or dimensions, that apply across different typographical spaces, so that it becomes possible to move smoothly from one typographical space to another.

Example Typographical Spaces (1)

- *Latin, Greek, Cyrillic, IPA*: left-to-right, discrete glyphs, numerous diacritics, stacked vertically, above or below the base letters, liberal use of hyphenation;
- *Hebrew*: right-to-left, discrete glyphs, optional use of diacritics (vowels and breathing marks), which are stacked horizontally below the base letter;
- *Arabic*: right-to-left, contiguous glyphs, contextually shaped, many ligatures, optional use of diacritics (vowels, breathing marks), placed in 1.5-dimensions, above or below;

Example Typographical Spaces (2)

- *Indic scripts*: left-to-right, 1.5-dimensional layout of clusters, numerous ligatures, applied selectively according to linguistic and stylistic criteria;
- *Chinese, Japanese*: vertical or left-to-right, often on fixed grid, with annotations to the right or above the main sequence of text, automatic word recognition needed for any form of analysis;
- *Egyptian hieroglyphics*: mixed left-to-right and right-to-left, 1.5-dimensional layout.

Conclusion

- Context-dependent user interface will be part of Ω_2.
- Context-dependent typesetter is current area of research.