A Multidimensional Approach to Typesetting

John Plaice

School of Computer Science and Engineering,
The University of New South Wales, Sydney, Australia
email: plaice@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~plaice/

Paul Swoboda

School of Computer Science and Engineering,
The University of New South Wales, Sydney, Australia
email: paul@omega.cse.unsw.edu.au

Yannis Haralambous

Département Informatique, École Nationale Supérieure des Télécommunications de Bretagne,
BP832, F-29285 Brest Cédex, France
email: Yannis.Haralambous@enst-bretagne.fr
http://omega.enstb.org/yannis/

Chris Rowley

Faculty of Mathematics and Computing, The Open University,
Milton Keynes, MK7 6AA, United Kingdom
email: C.A.Rowley@open.ac.uk

Abstract

We propose to create a new model for multilingual computerized typesetting, in which each of language, script, font and character is treated as a multidimensional entity, and all combine to form a multidimensional context. Typesetting is undertaken in a typographical space, and becomes a multiple-stage process of preparing the input stream for typesetting, segmenting the stream into clusters or words, typesetting these clusters, and then recombining them.

Each of the stages, including their respective algorithms, is dependent on the multidimensional context. This approach will support quality typesetting for a number of modern and ancient scripts. The paper and talk will show how these are to be implemented in Omega.

Introduction

We propose to create a radically new and practical model for character-level typesetting of all the world's languages and scripts, old and new. This goal is currently unattainable by any existing system, because of the underlying assumption that entities such as script, language, font, character and glyph are discrete, eternal and unchanging, as is supposed, for example, in the standards for Unicode [19], XML [20] and XSL [21]).

The key innovations in this proposal are (a) the assumption that these entities, their relationships and the processes (programs) applied to them are all arbitrarily parametrizable by a tree-structured context, and (b) the explicit manipulation of the complex and dynamic relationships between a (logical) character stream input and its visual representation on a particular medium as positioned glyphs.

These innovations lead directly to the concept of a *typographical space* that constrains the variance in the context and effectively embodies a certain set of processes and customs — as once might have been practiced in a typesetting workshop — while still allowing parametrization by the context.

Quality multilingual typesetting, as opposed to quality typesetting of unilingual documents for a number of different languages, requires the juxtaposition of separate typographical spaces. The separate spaces encourage the development of specialized algorithms to properly support widely different languages, scripts, and output substrates. The fact that the same tree-structured context permeates all of these different spaces ensures that key parameters can be shared across — or at least have correspondences in — different spaces, thereby ensuring consistency from one typographical space to another.

This approach greatly simplifies a number of tricky problems when one refers to language and script. Consider, for example, the English language: it is a multidimensional complex, varying through time (Old, Middle and Modern English), space (national and regional Englishes), and culture (science, arts, business, diplomacy, etc.). Understanding this variance is important: just as a simple example, US English and UK English have different spellings and hyphenations, and require different rules.

Scripts and their use have evolved similarly. In the past century, German and Turkish have both adopted the Latin script (from the Gothic and Arabic scripts, respectively). Chinese is printed, depending on the country, using ‘traditional’ or ‘simplified’ characters; transliteration for Chinese into the Latin script uses either the Wade-Giles or the *pinyin* method. This evolution and diversity means that documents encoded using one script should, possibly with the aid of linguistic tools, be printable using other scripts. For example, Ω, using a German morphological analyzer, can automatically print historical German texts in the Gothic script, even though they were encoded in the Latin script.

The relationship between character and glyph has also evolved, in inconsistent ways. A *character* is a unit of information exchange, while a *glyph* is a unit of visual information. If we consider, for example, the glyph *æ*, used in *mediæval*, it is considered in English to be a ligature — a variant glyph — for *ae*, while in Danish it is considered to be a character in its own right. In fact, one of the authors (Haralambous) [3] has shown that glyphs and characters are not absolutes, but, rather, are fluid concepts dependent on the context.

These relationships become more complex when we are faced with paleo-scripts from the mediæval and Ancient worlds. For example, there are something like 200 recognized Indic scripts, all derived from the Brahmi script. They all have similar — but clearly not identical — structure and there are situations in which it is natural to consider them as separate scripts while in other situations it is easier to consider them as variants of a single script.

We propose to use a tree-structured context to describe, to the desired level of precision, the entities that are being manipulated. This context will be used to describe a) how exactly to interpret the input; b) the exact format of the output; and c) the required processing. The latter should define how many passes over the input are required, what linguistic, layout or other plug-in tools should be used, along with the parametrization for each of these tools. An example context would be:

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

where `input` and `output` are called *dimensions*, and `lang:script` a *compound dimension*. The context will be inferred from environment variables, system locale, user profiles, command-line arguments, menu selections, and document markup.

This approach was first outlined in a position paper written by three of the authors [13], but at the time we had not understood the importance of the *typographical space*. It is the typographical space that allows us to fix exactly the meanings of character, glyph, language, script and font, in so doing facilitating the construction of modular and flexible typesetters, that allow automatic linguistic tools to add arbitrary markup to a text before it is printed, much as a traditional typographer might have used dictionaries and grammar books before pouring the lead.

To transform the above basic ideas into real, functional software usable for typesetting real, multilingual documents is not a trivial task. In this paper, we outline the steps that have led to the current ideas, and elaborate on problems still to be resolved.

We begin with a quick summary of the TEX character-level typesetter. Then we explain how the introduction of ΩTPs and ΩTP-lists in Ω provides a sophisticated means for adapting TEX's typesetter to multilingual typesetting. What Ω offers to the specialist user is great flexibility in manipulating the many different parameters needed for high-quality typesetting of different scripts.

However, this programming flexibility, with its large numbers of parameters, greatly complicates the user interface. The answer lies in being able to explicitly manipulate an *active run-time context* that permeates the entire typesetting process. We describe below how versioned macros, ΩTPs, and ΩTP-lists have been added to Ω to offer a high-level interface that a non-specialist user can manipulate with success.

Once such an active context is added to the interface, it becomes natural to incorporate the context into the entire process, and to completely redesign TEX's character-level typesetter. We examine below the initial proposal for such a typesetter, using the typographical space. We conclude by proposing a number of natural typographical spaces, along with their relevant parameters.

Computer Typesetting, TEX and Ω

For this paper, we define computer typesetting to be "*The production of printed matter by computer, ultimately to be viewed on some output medium*". The origins of computer typesetting go back to the 1950s, but it was not until 1982, with TEX [6], that it became possible to use computer software for high-quality typesetting of English and mathematics, as in *The Art of Computer Programming* [5].

At the character level, TEX can work in *text mode* or in *math mode*. In text-mode, *characters* in the input file are transformed almost directly into *glyphs* ('pictures' of characters) in the current font, and these glyphs are positioned side-by-side 'on the *baseline*'. A font-specific finite-state automaton can be used to change the glyphs used (by using *ligatures*) and their horizontal placement (by *kerning*). The 'words' thus typeset are then separated by a font-specific amount of stretchable inter-word space (*glue*) to form the stream of typeset glyphs that is passed to TEX's paragrapher. In math-mode, TEX uses a hand-crafted algorithm to lay out glyphs in *1.5* dimensions (this notation comes from frieze patterns).

The resulting stream of typeset glyphs is fed to TEX's *paragraphing algorithm* [7], which breaks the typeset stream for a paragraph at optimal — according to some acceptabilty criterion — places to produce lines of text placed in horizontal boxes. A much simpler algorithm is used for cutting pages from a continuous galley of such boxes. All computations in TEX are based on the width, height and depth of boxes, and these are derived ultimately from the same metrics for glyphs in the fonts.

The Ω system [10], developed by Plaice and Haralambous, is a series of extensions to the TEX system that facilitate multilingual typesetting. In Ω, the input character stream is processed by a series of filters, ultimately generating another character stream. Once all of the filters are applied, the resulting stream is passed to the TEX text-mode typesetter. We have written filters for character set conversion, transliteration, morphological analysis, spell-checking, contextual analysis, and 1.5-dimensional layout. The Ω system has been used to typeset alphabetic scripts from Europe and the Caucasus, cursive scripts from the Middle East, South Asia and South-East Asia, and East-Asian ideograms.

With the ΩTP mechanism, one can call many different filters for many different tasks. It often happens that some of these filters are only to be used in a selective manner, which very quickly creates a combinatorial explosion of new ΩTP-lists, hardly a favorable situation. This is resolved by introducing the run-time context of intensional programming, explained in the following sections.

Intensional Programming

Intensional programming [14] is a form of computing that supposes that there is a multidimensional context, and that all programs are capable of adapting themselves to this context. The context is pervasive, and can simultaneously affect the behavior of a program at the lowest, highest and middle layers.

When an intensional program is running, there is a *current context*. This context is initialized upon launching the program from the values of environment variables, from explicit parameters, and possibly from active context servers. The current context can be modified during execution, either explicitly through the program's actions, or implicitly, through changes at an active context server.

A context is a specific point in a multidimensional space, i.e., given a dimension, the context will return a value for that dimension. The simplest contexts are dictionaries (lists of attribute-value pairs). A natural generalization is what will be used in this paper: the values themselves can be contexts, resulting in a tree-structured context. The set of contexts is furnished with a partial order $\sqsubseteq$ called a *refinement relation*.

During execution, the current context can be queried, dimension by dimension, and the program can adapt its behavior accordingly. In addition, if the programming language supports it, then contextual conditional expressions and blocks can be defined, in which the *most relevant* case, with respect to the current context and according to the partial order, is chosen among the different possibilities.

In addition, any entity can be defined in multiple *versions*, (context, object) pairs. Whenever an identifier designating an entity appears in an expression or a statement, then the most relevant version of that entity, with respect to the current context, is chosen. This is called the *variant substructure principle*. The general approach is called *intensional versioning* [16].

The ISE programming language [15, 18] was the first language combining both intensional programming and versioning. It is based on the procedural scripting language Perl, and it has greatly facilitated the creation of multidimensional Web pages. Similar experimental work has been undertaken, under the supervision of the author Plaice, with C, C++, Java, and Eiffel. And, when combined with a context server (see Swoboda's PhD thesis [17]), it becomes possible for several documents or programs to be immersed in the same context.

Structuring the Context

We use the same notation to designate contexts and versions of entities. This section has three subsections. First, we define *contexts* and the refinement relation. Then, we define *version domains*, which hold versioned entities. Finally, we define *context operators*, which are used to change from context to context. In the following section, we will show how all of these are to be used.

Contexts and Refinement. Let $\left\{(\mathbb{S}_i, \sqsubseteq_i)\right\}_i$ be a collection of disjoint sets of ground values, each with its own partial order. Let $\mathbb{S} = \cup_i \mathbb{S}_i$. Then the set of contexts $\mathbb{C}$ ($\ni C$) over $\mathbb{S}$ is given by the following syntax:

$$
\begin{aligned}
C &::= \bot \mid A \mid \Omega \mid \langle B; L\rangle && (1)\\
B &::= \epsilon \mid \alpha \mid \omega \mid v && (2)\\
L &::= \emptyset \mid d{:}C + L && (3)
\end{aligned}
$$

where $d, v \in \mathbb{S}$.

There are three special contexts:

- $\bot$ is the *empty context* (also called *vanilla*);
- A is the *minimally defined context*, just more defined than the empty one;
- Ω is the *maximally defined context*, more defined than all other contexts.

The normal case is that there is a *base value* B, along with a *context list* (L for short), which is a set of *dimension-context* pairs. We write δL for the set of dimensions of L.

A sequence of dimensions is called a *compound dimension*. It can be used as a path into a context. Formally:

$$D = \cdot \mid d{:}D \tag{4}$$

If C is a context, $C(D)$ is the subtree of C whose root is reached by following the path D from the root of C:

$$
\begin{aligned}
C(\cdot) &= C && (5)\\
\langle B; d{:}C' + L\rangle\ (d{:}D) &= C'(D) && (6)
\end{aligned}
$$

As with contexts, there are three special base values:

- ϵ is the *empty base value*;
- α is the *minimally defined base value*, just more defined than the empty base value;
- ω is the *maximally defined base value*, more defined than all others.

The normal case is that a base value is simply a scalar.

To the set $\mathbb{C}$, we add an *equivalence* relation $\equiv$, and a *refinement* relation $\sqsubseteq$. We begin with the equivalence relation:

$$\bot \equiv \langle \epsilon; \emptyset \rangle \tag{7}$$
$$A \equiv \langle \alpha; \emptyset \rangle \tag{8}$$
$$\Omega \equiv \left\langle \omega; \sum_{d \in \mathbb{S}} d : \Omega \right\rangle \tag{9}$$
$$\frac{L_0 \equiv_L L_1}{\langle B; L_0 \rangle \sqsubseteq \langle B; L_1 \rangle} \tag{10}$$

Thus, $\bot$ and A are notational conveniences, while Ω cannot be reduced. The normal case supposes an equivalence relation $\equiv_L$ over context lists:

$$\emptyset \equiv_L d : \bot \tag{11}$$
$$d : \langle B; L + L' \rangle \equiv_L d : \big(\langle B; L \rangle + \langle B; L' \rangle \big) \tag{12}$$
$$L \equiv_L \emptyset + L \tag{13}$$
$$L \equiv_L L + L \tag{14}$$
$$L + L' \equiv_L L' + L \tag{15}$$
$$L + (L' + L'') \equiv_L (L + L') + L'' \tag{16}$$

The $+$ operator is idempotent, commutative, and associative. Now we can define the partial order over entire contexts:

$$\bot \sqsubseteq C \tag{17}$$
$$C \sqsubseteq \Omega \tag{18}$$
$$\frac{C \neq \bot}{A \sqsubseteq C} \tag{19}$$
$$\frac{C_0 \equiv C_1}{C_0 \sqsubseteq C_1} \tag{20}$$
$$\frac{B_0 \sqsubseteq_B B_1 \quad L_0 \sqsubseteq_L L_1}{\langle B_0; L_0 \rangle \sqsubseteq \langle B_1; L_1 \rangle} \tag{21}$$

which supposes a partial order $\sqsubseteq_B$ over base values:

$$\epsilon \sqsubseteq_B B \tag{22}$$
$$B \sqsubseteq_B B \tag{23}$$
$$B \sqsubseteq_B \omega \tag{24}$$
$$\frac{B \neq \epsilon}{\alpha \sqsubseteq_B B} \tag{25}$$
$$\frac{v_0, v_1 \in \mathbb{S}_i \quad v_0 \sqsubseteq_i v_1}{v_0 \sqsubseteq_B v_1} \tag{26}$$

The last rule states that if v_0 and v_1 belong to the same set $\mathbb{S}_i$ and are comparable according to the partial order $\sqsubseteq_i$, then that order is subsumed for refinement purposes.

The partial order over contexts also supposes a partial order $\sqsubseteq_L$ over context lists:

$$\emptyset \sqsubseteq_L L \tag{27}$$

$$\frac{L_0 \equiv_L L_1}{L_0 \sqsubseteq_L L_1} \tag{28}$$

$$\frac{C_0 \sqsubseteq C_1}{d{:}C_0 \sqsubseteq_L d{:}C_1} \tag{29}$$

$$\frac{L_0 \sqsubseteq_L L_1 \quad L_0' \sqsubseteq_L L_1'}{L_0 + L_0' \ \sqsubseteq_L \ L_1 + L_1'} \tag{30}$$

Rule 30 ensures that the + operator defines the least upper bound of two context lists.

Context and Version Domains. When doing intensional programming, we work with *sets* of contexts, called *context domains*, written $\mathcal{C}$. There is one operation on a context domain, namely the *best-fit*. Given a context domain $\mathcal{C}$ of existing contexts and a requested context C_{req}, the best-fit context is defined by:

$$\mathit{best}(\mathcal{C}, C_{\text{req}}) = \max\{C \in \mathcal{C} \mid C \sqsubseteq C_{\text{req}}\} \tag{31}$$

If the maximum does not exist, there is no best-fit context.

Typically, we will be versioning *something*, an object of some type. This is done using *versions*, simply (C, object) pairs. *Version domains* $\mathcal{V}$ then become functions mapping contexts to objects. The *best-fit object* in a version domain is given by:

$$\mathit{best}_O(\mathcal{V}, C_{\text{req}}) = \mathcal{V}(\mathit{best}(\text{dom}\ \mathcal{V}, C_{\text{req}})) \tag{32}$$

Context Operators. Context operators allow one to selectively *modify* contexts. Their syntax is similar to that of contexts.

$$\begin{aligned}
C_{\text{op}} &::= C \mid [P_{\text{op}}; B_{\text{op}}; L_{\text{op}}] && (33)\\
P_{\text{op}} &::= -- \mid E && (34)\\
B_{\text{op}} &::= - \mid \epsilon \mid B && (35)\\
L_{\text{op}} &::= \emptyset_{L_{\text{op}}} \mid d{:}C_{\text{op}} + L_{\text{op}} && (36)
\end{aligned}$$

A context operator is applied to a context to transform it into another context. (It can also be used to transform a context operator into another; see below.) The $-$ operator removes the current base value, while the $--$ operator in P_{op} is used to clear all dimensions not explicitly listed at that level.

Now we give the semantics for $C\ C_{\text{op}}$, the application of context operator C_{op} to context C:

$$\begin{aligned}
C_0\ C_1 &= C_1 && (37)\\
\Omega\ C_{\text{op}} &= \text{error} && (38)\\
\langle B; L\rangle\ [--; B_{\text{op}}; L_{\text{op}}] &= \Big\langle B; L\backslash(\delta L - \delta L_{\text{op}})\Big\rangle\ [E; B_{\text{op}}; L_{\text{op}}] && (39)\\
\langle B; L\rangle\ [E; B_{\text{op}}; L_{\text{op}}] &= \Big\langle (B\ B_{\text{op}}); (L\ L_{\text{op}})\Big\rangle && (40)
\end{aligned}$$

The general case consists of replacing the base value and replacing the context list. First, the base value:

$$\begin{aligned}
B\, - &= \epsilon && (41)\\
B\ \epsilon &= B && (42)\\
B_0\ B_1 &= B_1 && (43)
\end{aligned}$$

Now, the context list:

$$\begin{aligned}
L\ \emptyset_{L_{\text{op}}} &= L && (44)\\
(d{:}C + L)\ (d{:}C_{\text{op}} + L_{\text{op}}) &= d{:}(C\ C_{\text{op}}) + (L\ L_{\text{op}}) && (45)\\
L\ (d{:}C_{\text{op}} + L_{\text{op}}) &= d{:}(\bot\ C_{\text{op}}) + (L\ L_{\text{op}}), \quad d \notin \delta L && (46)
\end{aligned}$$

Context operators can also be applied to context operators. There are two cases:

$$\begin{aligned}
[P_{\text{op}}; B_{\text{op}0}; L_{\text{op}0}]\ [E; B_{\text{op}1}; L_{\text{op}1}] &= \Big[P_{\text{op}}; (B_{\text{op}0}\ B_{\text{op}1}); (L_{\text{op}0}\ L_{\text{op}1})\Big] && (47)\\
[P_{\text{op}}; B_{\text{op}0}; L_{\text{op}0}]\ [--; B_{\text{op}1}; L_{\text{op}1}] &= \Big[--; (B_{\text{op}0}\ B_{\text{op}1}); \big((L_{\text{op}0}\backslash(\delta L_{\text{op}0} - \delta L_{\text{op}1}))\ L_{\text{op}1}\big)\Big] && (48)
\end{aligned}$$

Now that we have given the formal syntax and semantics of contexts, version domains, and context operations, we can move on to typesetting.

The Running Context in Ω

As is standard, the abstract syntax is simpler than the concrete syntax, which offers richer possibilities to facilitate input.

Here is the concrete syntax for contexts in Ω:

C	::=	`<>`	Empty context
	|	`~~`	Minimum context
	|	`^^`	Maximum context
	|	`<`*val*`>`	Base value
	|	`<`*L*`>`	Subversions
	|	`<`*val*`+`*L*`>`	Base & subversions
val	::=	`~`	Minimum value
	|	`^`	Maximum value
	|	*string*	Normal value
L	::=	*dim*`:`C [`+` *dim*`:`C]*	
dim	::=	*string*	

Here is the concrete syntax for context operations:

C_{op}	::=	C	Replace the context
	|	`[]`	No change
	|	`[`val_{op}`]`	Change base
	|	`[`L_{op}`]`	Change subversions
	|	`[`val_{op}`+`L_{op}`]`	Change base & subs
val_{op}	::=	`-`	Clear base
	|	*val*	New value
	|	`--`	Clear subversions
	|	*val*`+--`	New base, clear subs
	|	`---`	Clear base & subs
L_{op}	::=	*dim*`:`C_{op} [`+` *dim*`:`C_{op}]*	

In Ω, the current context is given by:

`\contextshow{}`

If D is a compound dimension, then the subversion at dimension D is given by:

`\contextshow{`D`}`

while the base value at dimension D is given by:

`\contextbase{`D`}`

This context is initialized at the beginning of an Ω run with the values of environment variables and command-line parameters. Once it is set, it can be changed as follows:

`\contextset{`C_{op}`}`

Adapting to the Context

During execution, there are three mechanisms for Ω to modify its behavior with respect to the current context: (1) *versioned execution flow*, (2) *versioned macros*, and (3) *versioned* Ω*TPs*.

Execution Flow. The new `\contextchoice` primitive is used to change the execution flow:

`\contextchoice { {`C_{op_1}`}=>{`exp_1`}, ... {`C_{op_n}`}=>{`exp_n`} }`

Depending on the current context C, one of the expressions exp_i will be selected and expanded. The one chosen will correspond to the *best-fit* context among $\{C\ C_{\mathrm{op}_1},\ \ldots,\ C\ C_{\mathrm{op}_n}\}$ (see the discussion above of Context and Version Domains).

Macros. The Ω macro expansion process has been extended so that any control sequence can have multiple, *simultaneous* versions, at the same scoping level. Whenever \controlsequence is expanded, the *most relevant*, i.e. the *best-fit*, definition, with respect to the current context, is expanded.

A version of a control sequence is defined as follows:

`\vdef{`C_{op}`}`\controlsequence args{*definition*}

If the current context is C, then this definition defines the $C\ C_{\mathrm{op}}$ version of \controlsequence. The scoping of definitions is the same as for TeX.

This approach is upwardly compatible with the TeX macro expansion process. 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.

As stated above, during expansion the best-fit definition, with respect to the current context, of \controlsequence will be expanded whenever it is encountered. It is also possible to expand a particular version of a control sequence, by using:

`\vexp{`C_{op}`}`\controlsequence

ΩTPs and ΩTP-lists. Beyond the ability to manipulate larger data structures than does TEX, Ω allows the user to apply a series of filters to the input, each reading from standard input and writing to standard output. Each of the filters is called an ΩTP (Ω Translation Process), and a series of filters is called an ΩTP-list.

There are two kinds of ΩTP: internal and external. Internal ΩTPs are finite state machines written in an Ω-specific language, and they are compiled before being interpreted by the Ω engine. External ΩTPs are stand-alone programs, reading from standard input and writing to standard output, like Unix filters.

Internal and external ΩTPs handle context differently. For external ΩTPs, the context information can be passed on through an additional parameter to the system call invoking the external ΩTP:

program `-context=`*context*

Internal ΩTPs have been modified so that every instruction can be preceded by a context tag. Using the simplest syntax, this becomes:

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

When an internal ΩTP is being interpreted, an instruction is only examined if its context tag (defaulting to the empty context) is less than the current running context.

When ΩTPs and ΩTP-lists are being declared in Ω, the `\contextchoice` operator can be used to build versioned ΩTP-lists. With versioned ΩTP-lists, it becomes possible to define a single ΩTP-list with n ΩTPs, and each of the n ΩTPs can be activated with a separate parameter.

The versioned interface finally provides a user-level means for manipulating the large sets of parameters that must be handled when doing complex multilingual typesetting. When a transliterator is needed, the appropriate parameter is set. When a more complex layout mechanism is chosen, then another parameter is set. When spell-checking is desired, then another parameter is set. And so on. And the macros and ΩTP-lists adapt accordingly.

Because of the flexibility of the new interface, it is simpler to suppose that Ω always has an active ΩTP-list, and that it changes its behavior as the text changes its parameters. According to this vision then, multilingual typesetting simply means changing parameters as needed.

The versioned approach also resolves an issue that has been vexing the authors ever since the Ω and LATEX Projects have been trying to design a high-level interface for Ω usable by LATEX. The problem is that a language is not a monolithic, isolated, eternal and unchanging entity. 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.

Context-Dependent Typesetting

The existing Ω framework is very powerful, in the sense that the ΩTPs can make the TEX character-level typesetter stand on its head to produce amazing results, without the end-user having to know what is going on. However, it is hardly a natural process to take a character-level typesetter designed for English with its isolated glyphs and occasional ligatures and then to use it to undertake complex Arabic typesetting with its numerous ligatures and floating diacritics.

Far more appropriate is to break up the typesetting process into separate modules, and to parameterize each of these with the current context.

In the most general sense, a typesetter is a program that transforms a stream of characters into a stream of positioned glyphs. We can separate out three themes:

- *Atomic typesetting* is the transformation of a (small) fully marked-up stream of characters into a stream of positioned glyphs. An atomic typesetter might be used directly by an application that prints one or two words at different points on a computer screen, e.g. by mapping software to print out a city or river name, or by a more complex continuous typesetter.

- *Continuous typesetting* is the transformation of a (larger) stream of characters into a stream of positioned glyphs that can be segmented at different points to produce several lines (or other structures) of typeset text.
- *Preparing the input* is the process of applying several programs to a stream of characters to add additional markup so that the typesetter can fully do its work.

A continuous typesetter would typically use one or more atomic typesetters, and might also require input to be prepared.

Below, we give a simple model of a continous typesetter. It is split into four separate *phases*: *preparation*, *segmentation*, *micro-typesetting* and *recombination*. Each of these phases is dependent on the context, and we write the process, using C++ syntax, as:

```
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;
}
```

where *function*`.best(`*context*`)` means that the most relevant version of *function*, with respect to *context*, is selected. We examine each of the phases in detail.

Preparation.

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

The preparation phase in this new approach is similar to the current situation in the Ω system. At all times, there is an active Ω Translation Processing List (ΩTP-list). This list consists of individual Ω Translation Processes (ΩTP's), each of which is a filter reading from standard input to standard output. What is new is that the whole process becomes context-dependent. First, the most relevant ΩTP-list, with respect to the context and using the refinement relation over contexts, is the one that is active. Second, once chosen, it can test the current context and adapt its behavior, by selectively turning on or off, or even replacing, individual ΩTP's.

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. In English, word detection is a trivial problem, and segmentation just means recognizing 'white space' such as the blank character, Unicode `U+020`. By contrast, in Thai, where there is normally no word-delimiter in the character stream (blanks are traditionally only used as sentence-delimiters), it is impossible to do any form of automatic processing unless a sophisticated morphological analyzer is being used to calculate word and syllable boundaries. In many Germanic and Slavic languages, it is also necessary to find the division of compound words into their building blocks. These processes are closely related to finding word-division points so this should be incorporated into this part of the process (a very different approach to that of TEX). The choice of segmenter is thus clearly seen to be context-dependent.

Cluster typesetting.

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

During the typesetting phase, a *cluster engine* processes a character cluster, taking into account the current context including language and font information, and produces the typeset output — a sequence of positioned glyphs. In many cases, such as 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. When dealing with complex scripts or fonts allowing great versatility (as with Adobe Type 3 fonts), many different cluster engines are needed: these are selected and their behaviour is fine-tuned according to the context.

Recombination.

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

The final phase, before calling a higher-level formatting process such as a paragrapher, is the recombination phase. Here, the typeset clusters are placed next to each other. For simple text, such as the English in this proposal, 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.

Typographical Spaces

Given the sophistication of the multiple-phase process, and that the choice of segmenter, cluster engine and recombiner are all context-dependent, and that the actions of each of these, once they are chosen, also depends on the context, this new model of typesetting engine is potentially *much* more powerful than anything previously proposed or implemented. However, there remains a key problem in the type of the function:

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

In this type declaration, the types `Glyph` and `Char` appear to be normal datatypes, i.e., fixed, unchanging sets, which is not at all consistent with our view that character and glyph should be perceived as multidimensional entities.

Really, the sets for character and glyph should be context-dependent. However, if these basic types were to continually change, then it would be very difficult to write any of the algorithms, because one could never be sure of the ultimate particles, the atoms, with which one was working.

It is to resolve this problem that we introduce the *typographical space*. This space is designed to constrain the variance in the context. Within a specific typographical space, the types for character and glyph remain fixed. Hence the above type becomes something like:

```
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 as appropriate by the algorithms within that space.

Suppose there was a typographical space for Greek typesetting, including modern and ancient Greek, literary Greek and colloquial Greek, as well as other languages that have been typeset using the Greek alphabet. Then the character datatype would most likely correspond to a subset of Unicode, augmented by additional characters that were not included in the standard. The glyph datatype would consist of many glyphs, and could contain a number of precomposed multi-accented glyphs or a smaller set of isolated glyphs, including accents, that are to be placed at appropriate places.

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 languagest that they know best. 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 Spaces

We intend to test and validate the model described above by creating, at least, typographical spaces for the following scripts:

- *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 and breathing marks), placed in 1.5-dimensions, above and below;
- *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 — in Chinese and Japanese, "words" use one or more characters, but these are not visually apparent — needed for any form of analysis;
- *Egyptian hieroglyphics*: mixed left-to-right and right-to-left, 1.5-dimensional layout.

Once these basic spaces are validated, then further experiments, viewing language as a multidimensional entity, can be undertaken. Already with Ω, we have typeset Spanish with both the Hebrew and Latin scripts; Berber with the Tifinagh, Arabic and Latin scripts; Arabic with Arabic, Hebrew, Syriac, Latin and even *Arabized Latin* (Latin script with a few additional glyphs reminiscent of the Arabic script). The Arabic script can be rendered in Naskh or Nastaliq or many other styles. Japanese can be typeset with or without *furigana*, little annotations above the *kanji* (the Chinese characters) to facilitate pronunciation. Some of the corresponding typographical spaces will be quite interesting.

The objective is to incorporate solutions to all such problems, currently solved in an *ad hoc* manner, into our framework; each time, the key is to correctly summarize the typographical space. With this key, then the choice of segmenters, clusters engines and recombiners to build, and of how they are built, is clarified; nevertheless, these algorithms may remain complex, because of the inherent complexity of the problems they are solving.

Conclusions

When we have fully developed this model, we will be able to produce, with relative ease, high-quality documents in many different languages and scripts.

Furthermore, this new approach of using contexts can be used to improve not just micro- but also macro-typesetting. Rowley, as one of the leaders of the LaTeX3 Project, has worked with closely related ideas in the context of Mittelbach's *templates* for higher-level formatting processes [2]. Here the particular instance of a template object that is used to format a document element will depend on a context that is derived from both the logical position of that element in the structured document and from the formatting of the physically surrounding objects in the formatted document. Collaboration between the current authors and other members of the LaTeX3 team will lead to many new interfaces that give access to the new functionality.

Other examples of the importance of such a structured context in document processing can be found in work by Rowley with Frank Mittelbach [9].

Another example of dependence on this visual context occurs in the use of Adobe Type 3 fonts, which are designed so that glyphs can be generated differently upon each rendering (see [1] for a discussion of a number of effects). On another level, the Open-Type standard for font resources [11] allows for many different kinds of parameters — beyond the basic three of width, height, and depth —, multiple baselines, and a much richer notion of ligature. Our new engine for micro-typography will provide new capabilities, adaptable to new kinds of parameters, and increased control. Thus we shall be able to provide a simple high-level interface that takes advantage of new developments in font technologies.

Finally, this proposed model should be understood as the preparation for a much more ambitious project, that will deal not just with low-level typesetting but also general problems of document structuring and layout for demanding typographic designs in a highly automated environment. Detailed discussion along these lines has already been initiated between the Ω and LaTeX3 projects, which look forward to these wider horizons.

References

[1] Jacques André. *Création de fontes en typographie numérique*. Documents d'habilitation, IRISA+IFSIC, Rennes, 1993.

[2] David Carlisle, Frank Mittelbach and Chris Rowley. New interfaces for LaTeX class design, 1999. http://www.latex-project.org/papers/tug99.pdf

[3] Yannis Haralambous. Unicode et typographie : un amour impossible. *Document numérique* 6(3–4):105–137, 2002.

[4] P. Karow. *hz-Programm, Mikrotypographie für den anspruchsvollen Satz*. Gutenberg-Jahrbuch, Mainz, 1993.

[5] D. E. Knuth. *The Art of Computer Programming*. 3 vol., Third Ed., Addison-Wesley, 1997.

[6] D. E. Knuth. *Computers and Typesetting*. 5 vol., Addison-Wesley, 1986.

[7] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software – Practice and Experience* 11(11):1119–1184, 1981.

[8] Frank Mittelbach and Chris Rowley, 1996. Application-independent representation of text for document processing. http://www.latex-project.org/papers/unicode5.pdf

[9] Frank Mittelbach and Chris Rowley. Language information in structured documents, 1997. http://www.latex-project.org/papers/language-tug97-paper-revised.pdf

[10] Omega Typesetting and Document Processing System. http://omega.cse.unsw.edu.au

[11] OpenType. http://www.opentype.org

[12] John Plaice and Yannis Haralambous. Generating multiple outputs from Omega. *TUGboat*, 2003. To appear.

[13] John Plaice, Yannis Haralambous and Chris Rowley. An extensible approach to high-quality multilingual typesetting. In *RIDE-MLIM* 2003, IEEE Computer Society Press, 2003.

[14] John Plaice and Joey Paquet. Introduction to intensional programming. In *Intensional Programming I*, World-Scientific, Singapore, 1996.

[15] John Plaice, Paul Swoboda and Ammar Alammar. Building intensional communities using shared contexts. In *Distributed Communities on the Web*, *LNCS* 1830:55–64, Springer-Verlag, 2000.

[16] John Plaice and William W. Wadge. A new approach to version control. *IEEE-TSE* 19(3):268–276, 1993.

[17] Paul Swoboda. *A Formalization and Implementation of Distributed Intensional Programming*. PhD Thesis, The University of New South Wales, Sydney, Australia, 2003.

[18] Paul Swoboda. *Practical Languages for Intensional Programming*. MSc Thesis, University of Victoria, Canada, 1999.

[19] Unicode Home Page. http://www.unicode.org

[20] Extensible Markup Language (XML). http://www.w3c.org/XML

[21] The Extensible Stylesheet Language (XSL). http://www.w3c.org/Style/XSL