Supporting multidimensional documents with Omega*

John Plaice† and Yannis Haralambous‡

Preliminary — 25 February 2001

Abstract

We propose to add multidimensionality to the Omega Typesetting and Document Processing System. The identifiers — control sequences, fonts and input files — of an Omega document will be allowed to vary through a multiparametric or multidimensional space. By changing the values associated with a set of parameters, the expressions associated with those identifiers may change, and the behavior of those expressions may also adapt.

Using this model, called *intensional*, will simplify the creation of macros for multilingual computing and for fonts with many parameters. In addition, it will allow macro packages that can do both typesetting and markup generation, as needed.

1 Introduction

The constantly changing landscape of the computing infrastructure is forcing us to develop an understanding of programming in a context. *Intensional programming* is a general approach to the building of programs whose behavior depends on an implicit multidimensional context.

An intensional context can be probed, and changed explicitly, dimension by dimension, in order to change the control flow. Contexts are first-class values, which means that the entire context can be replaced, cloned, split or shared with other programs.

When an identifier (variable, macro, function, etc.) is encountered in an intensional program, then the most relevant definition for that identifier (the best-fit version) is chosen. This definition is then evaluated or executed in the current context or in some other explicitly mentioned context.

At the semantic level, an intensional program is considered to be a function from the set of all possible contexts (in the terminology of *intensional logic*, the set of all *possible worlds*), to the set of all possible (ordinary) programs.

Intensional languages have been built by taking existing languages and by adding context-manipulation primitives. This has been done for functional programming languages (ISWIM + intensionality → Lucid), for scripting languages (Perl + intensionality → ISE) and for markup languages (HTML + intensionality → Intensional HTML, XML + intensionality → Multidimensional XML).

Intensional programming is a general model that can be applied to deal with variance in operating system kernels, file systems networks, applications, sessions, etc. It is most effective when the name space – and the semantics – of the current context is shared by all parts of a system, but there is clearly room for local name spaces, a subject of current research.

When a new problem is encountered in an existing intensional program or set of programs, the typical response is to introduce a new dimension. Since the current context is rarely enumerated explictly, only those sections of code that are affected by the new dimension need be changed; this approach, used with great success with differential equations, ensures that code grows much more slowly than with most forms of programming.

*Presented to the *Fourth International Symposium on Multilingual Computing*, March 1-3, Tokyo, Japan.

†John Plaice, School of Computer Science and Engineering, The University of New South Wales, UNSW SYDNEY NSW 2052, Australia. plaice@cse.unsw.edu.au

‡Yannis Haralambous, Atelier Fluxus-Virus, 187 rue Nationale, F-59800 Lille, France. yannis@fluxus-virus.com

There is a Web site on intensional programming which is maintained by the first author (Plaice), at `http://www.cse.unsw.edu.au/~plaice/intensional/`. Two example intensional Web sites are `http://www.cse.unsw.edu.au/~plaice/louvre/`, which is a multidimensional visit to the Louvre Museum in Paris, and `http://www.cse.unsw.edu.au/~plaice/mcluhan/`, which is a collection of quotes by Marshall McLuhan, organized as poptext. In each case, the entire Web site is a single Web page, which reorganizes itself as the current context changes. The different parts that contribute to the page all adapt as needed to the current context.

We propose to take the same approach to Omega documents, thereby making Omega documents to be significantly more versatile than the current TEX documents.

2 The version space

Whatever kinds of objects or programs we are interested in, they are all supposed to vary within a *version space*, whose grammar is given below:

$$\begin{array}{rcl} \mathit{version}\ (V) & ::= & A \mid \Omega \mid \mathit{base}\ (+\ \mathit{dimension} : \mathit{version})^* \\ \mathit{base}\ (b) & ::= & \alpha \mid \omega \mid \epsilon \mid \mathit{scalar} \\ \mathit{dimension}\ (d) & ::= & \mathit{scalar} \\ \mathit{scalar}\ (s) & ::= & \mathit{id} \mid n \end{array}$$

where

A is the least defined version;

Ω is the totally defined version;

ϵ is the empty base value;

α is the least defined base value;

ω is the totally defined base value.

It is understood that some elements of the space are refinements of other elements of the same space. This refinement relation defines a partial order, given by the following equations:

$$\begin{array}{lll} \epsilon \sqsubseteq b & V \sqsubseteq \Omega & V : \epsilon = V \\ b \sqsubseteq b & V \sqsubseteq V : V' & \epsilon + V = V \\ b \sqsubseteq \omega & V \sqsubseteq V + V' & V + V = V \\ \dfrac{\epsilon \neq b}{\alpha \sqsubseteq b} & \dfrac{\epsilon \neq V}{A \sqsubseteq V} & V + V' = V' + V \\ & & V + (V' + V'') = (V + V') + V'' \\ & & V : (V' + V'') = V : V' + V : V'' \\ \dfrac{m \leq n}{m \sqsubseteq n} & \dfrac{V \sqsubseteq V' \quad W \sqsubseteq W'}{V + W \sqsubseteq V' + W'} & \end{array}$$

Suppose for example, that we wanted to encode the language and the script. Then we could write

`language:French+script:Latin` for French written in the Latin script.

`language:Arabic+script:Arabic` for Arabic written in the Arabic script.

`language:Arabic+script:Latin+transliteration:Omega` for Arabic written in the Latin script using the Omega transliteration.

`language:Arabic+script:Latin` for Arabic written in the Latin script using the standard transliteration.

`language:Berber+script:Latin` for Berber written in the Latin script.

`language:Persian+script:Arabic+style:Nastaliq` for Persian written in Nastaliq.

Now suppose that we wanted to encode dialect differences. Then we could write

`language:(French+dialect:Québec)+script:Latin` for Québec French.

`language:(French+dialect:(Québec+region:Gaspésie))+script:Latin` for the French of the Gaspésie region.

We say that `French` is the *base value* of `language`, while the *full value* of `language` is `French+dialect:(Québec+region:Gaspésie)`. We call `language` and `language:dialect` *dimensions*. So, as we can see, the space includes both multidimensionality and nesting. One can think of a multidimensional spreadsheet, where any cell can have as value a whole new multidimensional spreadsheet.

We write `language:French` $\sqsubseteq$ `language:(French+dialect:Québec)`, which means the latter is a *refinement* of the former.

3 Versioned macros

Control sequences — the macro identifiers — in TEX and Omega consist of an escape delimiter followed by a sequence of letters, as in `\macro`. To allow for versions of macros, we allow a version modifier to appear between the escape delimiter and the sequence of letters. Version modifiers are designated through the use of two new catcodes, *begin version modifier* and *end version modifier*, which we will designate in the rest of the document as < and >. For example, `\<language:French>chaptername` would be French language version of the `\chaptername` macro.

It is assumed that at all times, a program is running with a *current version*, which is simply a point in the version space. The current version can be changed *absolutely*, using the <–> pair, or relatively, using the [–] pair (new catcodes again).

Running `\vset<language:Arabic+script:Arabic>` will replace the current version with version `language:Arabic+script:Arabic`. From there, running `\vset[script:Latin]` will modify the current version to `language:Arabic+script:Latin`. The use of a single colon (':') replaces only the base value.

If `\vset<language:(French+dialect:(Québec+region:Gaspésie))+script:Latin>` is encountered, we get a new version. Then running `\vset[language::Arabic]` changes the current version to `language:Arabic+script:Arabic`. The use of the double colon ('::') replaces the full value.

For a given dimension *dim*, we can write `\the\vbase<`*dim*`>` or `\the\vfull<`*dim*`>` to access the base or full values of *dim* in the current version.

When a control sequence is encountered, then all of the current versions of the control sequence are examined, in order to find the *best-fit* version, which is the maximum of all of those versions less than or equal to the current version. That version is then chosen, and we continue.

Except... you can play with the versions when actually using a control sequence! Hence `\<`*requested*`><`*running*`>`*macro* looks for the best-fit to the *requested* version *macro*, as opposed to the current version, then runs the body of *macro* under the *running* version (itself interpreted within the current version).

With two more new catcodes, `?` and `!`, we can write `\?<`*requested*`><`*running*`>`*macro*, in which case the *running* version is interpreted according to the requested version, not the current version. We can also write `\!<`*requested*`><`*running*`>`*macro*, in which case the *running* version is interpreted according to the best-fit version, not the current version.

4 Versioned fonts

Versioned fonts would allow us to deal with problems of glyph changes when writing in different directions, with the different parameters associated with PostScript fonts, and with the successive version of fonts released by foundries, etc. We are currently working on developing font metrics that would allow us to deal with these complexities.

Currently fonts are defined as `\font` *name=filename*. To access versioned fonts, we need to add a version qualifier `\font` *name=<version>filename*.

5 Versioned files

If the file system itself or the interface to the file system, such as `kpathsea`, understands the version language, then we can also have versioned input files. The most natural mechanism is to do this at the file system level, but then would make any solution non-portable. We still need to study this problem a bit more before choosing a solution.

Currently files are read as `\input` *filename*. To access versioned files, we need to add a version qualifier `\input`*<version>filename*.

6 Conclusion

Addding multidimensionality to Omega will tremendously simplify the programming of language-dependent macros, as well as the creation of adaptable documents. More details are forthcoming as all this is implemented.