$\mathcal{NTS}$: a New Typesetting System

Karel Skoupý
Faculty of Informatics
Botanická 68a
602 00 Brno, Czech Republic
Phone: +420-5-752-040
skoupy@informatics.muni.cz

Abstract

$\mathcal{NTS}$ represents one possible radical approach to the idea of making a successor for TEX. Its underlying them is the complete re-implementation of TEX: The Program in Java. The first version will be compatible with TEX but the structure of the new program will be as open and modular as possible. At the time when TEX: The Program was written, computer performance and programming technology were very limited in comparison with today. Object orientation and the many other modern features of Java will make many problems easier to solve and will allow for far greater generality. Polymorphic objects will handle different font or output formats directly without affecting the rest of the system. The new implementation will provide a platform on which further experimentation can be conducted; such experimentation may aim, for example, to improve typographic quality (e.g. page break optimization) and/or facilitate integration with other systems.

When considering the future of TEX and its potential successor(s), there were five options available [9]. They ranged from the most conservative — to leave TEX exactly as it is — to the most radical — to design quite a new typesetting system for the next century. The $\mathcal{NTS}$ project started with a relatively conservative approach to add some extensions and enhancements to the current TEX which resulted in ε-TEX. It was agreed that the design of a new system from scratch is worthy but it had to be postponed until the $\mathcal{NTS}$ project had adequate financial resources.

The funds allocated to the $\mathcal{NTS}$ project by DANTE e.V. enabled the radical approach to $\mathcal{NTS}$ to get off the ground. The first meeting of the $\mathcal{NTS}$ group was held in Zeuthen (during a regular DANTE meeting) between October 8–11 1997. Although the current author (who was to be employed as a full-time programmer) was busy with another project, it was planned that he could start work on $\mathcal{NTS}$ near the beginning of 1998. It of course did not prevent the working group from thinking about the problem.

Fundamental Desiderata

It had been already decided that $\mathcal{NTS}$ should not be designed from scratch entirely — TEX: The Program (or more precisely ε-TEX) was chosen as the starting point. What had to be done from scratch was a complete re-implementation. Such a re-implementation has to be compatible with current TEX, and ideally it should pass the TRIP test (unless there are really good reasons for not passing it). This constraint has its advantages. As TEX is considered (by the working group) to be the best typesetting system on the world, compatibility will prove that $\mathcal{NTS}$ is at least as good as its predecessor. Also TEX: The Program is an extremely well designed application full of inspiring ideas and it would not be wise to drop them.

On the other hand the structure of the new system should be made as open as possible. It should be partitioned into relatively independent modules with well defined interface. It will eventually allow for great changes by only local intervention to a particular module without affecting the rest of the system. Although the functionality will be compatible with TEX, both the structure of the program and the data structures should be ready for such a changes and extensions that are likely to be desirable for $\mathcal{NTS}$ (Properties of a potential new typesetting system were discussed in [7, 3, 2]). For this purpose an object oriented design seemed to be the most suitable.

Implementation language The first task of the group was to choose a programming language for the implementation. Several high level languages

for fast prototyping had been taken into account in the past. Eventually three well known object oriented languages were considered: Common Lisp Object System (CLOS), C++ and Java. Although each of these languages has its specific advantages, after careful comparison of them Java was chosen.

Some of its characteristics follow. Java as the youngest of these languages could learn from them. It has a very advanced implementation of objects and their interfaces are orthogonal to the class hierarchy. Classes are compiled into so-called byte code and can be dynamically combined at run time. The built in garbage collector is very convenient and prevents the programmer from causing many memory violation errors. Types and their checking can also catch many errors, from trivial to serious. Exception handling requires precise declarations which prevent forgetting of boundary situations. Java is completely portable even at the level of compiled byte code. Although the standard interpretation by the Java Virtual Machine is not as efficient as machine code, there are compilers to native code available. There is also a wide range of standard libraries for networking, graphic, graphical user interfaces and others.

One very important point is Java's Internet awareness. You can imagine that you will be able to (automatically) download new modules and $\mathcal{NTS}$ plug-ins, share fonts and input texts over the Internet and use JavaBeans [1] to tailor the system for your needs interactively. We also anticipate good support for Java in future. This is now very popular and many new applications are being implemented in it. It is quite possible that a Java interpreter will eventually be burned onto a silicon chip at some point in the not-too distant future.

Design of $\mathcal{NTS}$

During the Spring of 1998 there were two more meetings of the $\mathcal{NTS}$ working group. Mainly discussed were the features of $\mathcal{NTS}$ which will not be implemented in the first version but which the design has to anticipate. Gradually the specification for the first version was set.

The first task of real work on the implementation was to make a design with proposed structure of the $\mathcal{NTS}$. It was ready by May 2 1998 and reviewed by Philip Taylor and Jiří Zlatuška. The presentation of the main design decisions will follow. We do not want to go into very implementation-specific details, it will instead be a rather informal description of things which might be of interest. Comments are of course welcome.

General notes The specification for re-implementation is rather simple. The main source of information is TEX: The Program itself. The task is not to design something with a different philosophy, it is rather to take used principles and make them more open and general. It seems that there were two main constraints at the time of the creation of TEX — the low performance of machines and the programming technology available.

The first problem related to these constraints is memory management. On the one hand the memories of computers at that time were very small compared with today, whilst on the other hand there was probably no standard support for dynamic memory management. Knuth decided to create his own memory management based on preallocated buffers. Today we are much less constrained in using memory. The physical memories of today's computers are much bigger and current operating systems provide even larger virtual memories.

The second problem was concerned with the data structures. It is apparent from the source code that Knuth tried to use memory in the most effective way. He did not accept the standard Pascal records and pointers and rather used preallocated arrays in a very compact way. This was good for performance but it would be very painful to add new data structures to existing scheme. The symbolic names for structure items are maintained using `WEB` macros and therefore their types are not distinguished and checked by compiler.

Fortunately Java provides us with very advanced memory management with garbage collection. It is very natural to accept Java objects as data structures, not to take special care about memory at all and not to impose any explicit limits to the size of internal buffers.

`WEB` preprocessor We believe that Knuth took the best programming language available for his purpose at the time. But even standard Pascal was not enough and he made the `WEB` preprocessor mainly for three reasons: literate programming, macro definitions and rearranging the source text.

Pascal serves well for the structured programming paradigm. But if you peek into TEX: The Program you can recognize steps towards an object oriented paradigm supported by `WEB`. The code dealing with particular data structures — mainly the relevant parts of switches in general procedures as (for example) symbolic printing or of course the `main_control` — is usually gathered in one place. In an

[1] JavaBeans is a component architecture that helps independent vendors write classes that can be treated as components of larger systems assembled by users.

object oriented language such processing is implemented by virtual methods of objects and there is no need for rearranging the order of source code and for most of the switches any more.

The need for macro definitions is very much reduced, too. In C++ the usage of macros was significantly replaced by templates. Unfortunately standard Java does not provide any of these facilities (there are other implementations which have templates and operator overloading as an extension, e.g. jump[2]). This should not impact too badly on $\mathcal{NTS}$: templates are mostly used in general frameworks.

We certainly want the new program to be well documented, but we are still not concerned to make a book of it. Java has its own source documentation facility. The documentation is in the form of pragmatic comments and is supposed to be in HTML or a similar SGML-based format. It seems sufficient to us now. Well, it may change in future but in such a case it is not necessary to change the program code itself.

Character set, fonts and hyphenation. In current TEX the input encoding, hyphenation pattern encoding and font encoding must be the same. Also there are only 256 character codes. It may be sufficient for English but it makes usage of TEX harder for other languages. There are ways around it (virtual fonts, pattern sources independent of character encoding), the thing just can be made much easier.

Fonts and hyphenation tables will be implemented as object providing methods under a well defined interface. It is possible that objects for different formats provide the same interface and for example PostScript font metrics can be used directly. We can make the interface to such objects independent of encoding using character names. Internally the characters will be represented by numbers but there will be a module for mapping to external names. In case the font does not know the names (pk fonts), the process will default to numeric codes. The codes can possibly be re-mapped by tables for different encodings and independent of fonts.

Diagnostics. TEX uses terminal and log files for diagnostic purposes. It can be made much more general by polymorphic log objects. Some users might prefer some windowed output (strange but possible), but mainly it can be used by some wrapping application having $\mathcal{NTS}$ as a processor inside.

Basic types. Numbers, dimens and glues will be in $\mathcal{NTS}$ as well. Their semantics will be the same as in TEX. Maybe in future we will add other types for some extended typesetting tasks.

Language. In TEX the input characters are transformed to tokens and they are after macro expansion transformed to primitive commands which are handled by the chief executive. In $\mathcal{NTS}$ the process will be similar but our aim is to separate different layers as much as possible. The input, tokenization and macro expansion will be a straightforward reimplementation using objects.

Although in TEX there are dependencies between the macro language and typesetting (the dimensions of boxes in registers, named typesetting parameters, output routine) we will try to make these dependencies clear and handled via well defined interface. This might allow us to provide sometime in the future an alternative input language (procedural, object oriented, ...) without any incompatible change (extensions might be necessary) to the typesetting engine driven by primitive commands.

Modes. The meaning of a primitive command may depend on the current mode. There are three main modes (vertical, horizontal, maths) in TEX each with two submodes. In $\mathcal{NTS}$ it will be much more general. The modes will not be just codes but polymorphic objects. They will provide methods such as adding the next character, kern or glue and so on. In some cases the method will issue an error message (`"You can't ... in ... mode"`), in other cases it will push another object onto the stack of modes. Each command will know how to apply itself when meeting with a mode.

This will allow for making new specialized modes derived from existing ones. We will try (for example) to implement alignments in this way. Commands not aware of any specialization will pass in the usual way but specialized commands can test the current mode and invoke some extra method not included in the base mode interface. This approach may make non-textual modes for chemical formulæ, pictures or music notes more easy and efficient.

Lists and boxes. The objects in lists will be polymorphic so the formatting algorithms will handle each object uniformly. They will invoke only the appropriate methods to get information about size, stretchability, ...

In TEX the lists are static once created. After breaking a paragraph into lines there is no easy way to reformat it. Also the parameters used by the paragraph breaking algorithm (such as `\tolerance` or `\hyphenpenalty`) are lost after processing. In

[2] jump is a free Java compiler with mentioned capabilities, see: `http://ourworld.compuserve.com/homepages/DeHoeffner/jump.htm`

$\mathcal{NTS}$ the lists will remain dynamic and keep all the information necessary for reformatting later.

One application might be a WYSIWYG interactive program which needs to reformat a paragraph after a user's change to it. We do not plan such a program to be part of $\mathcal{NTS}$ but $\mathcal{NTS}$ might be used by someone else as an internal engine and we will try to make it possible. Provided that the modules of $\mathcal{NTS}$ are independent, they can be used by such applications directly. In such case the applications can also employ lists as their own data structures.

But there is a more important reason from the point of view of high quality typesetting — global page break optimization. Actually this was one of the main challenges to $\mathcal{NTS}$. For solving this problem we will need to keep the whole main vertical list and try to find a satisfactory (or optimal) sequence of page breaks. Then reformatting of paragraphs might be helpful. If we allow the page layout (`\hsize`) to change for different pages it will become a necessity. Even without global optimization the changing page layout is a problem and more general shapes of paragraphs (than defined by `\hsize` and `\parshape`) will be useful.

Maths formulæ. TEX is excellent in the typesetting of mathematics. The formulæ have their own representation which is transformed to usual boxes when contributed to the parent list. The difference in $\mathcal{NTS}$ approach is that the objects for formulæ will be descendants of the same base class as the ordinary text. It will not need transformation and will be kept dynamically in the same way as lists.

Output. You might guess that also the output will be shipped out by a polymorphic object. It will have methods for setting characters, rules, images or some graphic objects. The objects in the list will know which method to invoke. One implementation can produce DVI, others can generate PostScript or PDF.

Algorithm parameterization. TEX's algorithms are parametrized in many ways. There are a lot of numeric parameters to the paragraph breaking algorithm, page output and page breaking is influenced by a user defined output routine. We can make parameterization more general. Beside an enriched set of variable parameters we will prepare void virtual methods which can (for example) add extra demerits to two consecutive broken lines in a paragraph or to a whole sequence of potential line breaks. By supplying some smart code to such methods, it will be possible to avoid "rivers" and other subtleties not solved by TEX. Such parameterization will be done by making a specialized version of $\mathcal{NTS}$ with overridden methods or by more convenient plug-ins. The possibility of giving access to internal lists in the input language and user supplied methods from the input file should be discussed.

Conclusion

We described the current state of the initial phase of the $\mathcal{NTS}$ project aiming at re-implementing TEX in Java so that the internal structure of the program will allow for experiments and modifications of the algorithms used or the actions taken when typesetting using TEX. Basic design decisions behind the choice of the implementation language and the object-oriented programming paradigm have been exposed and the overall structure of the resulting program has been outlined.

The first version of the $\mathcal{NTS}$ is now under development and should be available by the beginning of 1999.

The author wishes to express thanks to Don Knuth for making TEX available, the $\mathcal{NTS}$ group for fruitful discussions, contributors to the `NTS-L` list for many interesting ideas, and DANTE e.V. for continuing support in this endeavor.

References

[1] Ken Arnold, James Gosling: "The Java Programming Language, Second Edition", Addison-Wesley Publishing Company, Reading, Mass., December 1997.

[2] Michael Barr: "TEX wish list", in *TUGboat*, Vol. 13, No. 2, pp. 223–226, July 1992.

[3] Nelson H. F. Beebe: "Comments on the future of TEX and METAFONT", in *TUGboat*, Vol. 11, No. 4, pp. 490–494, November 1990.

[4] Roger Hunter: "A future for TEX", in *TUGboat*, Vol. 14, No. 3, pp. 183–186, October 1993.

[5] Donald E. Knuth: "The future of TEX and METAFONT", in *TUGboat*, Vol. 11, No. 4, pp. 489–489, November 1990.

[6] Donald E. Knuth: "TEX: The Program", Addison-Wesley Publishing Company, Reading, Mass., 1986.

[7] Frank Mittelbach: "E-TEX: Guidelines for future TEX", in *TUGboat*, Vol. 11, No. 3, pp. 337–345, September 1990.

[8] Philip Taylor: "TEX: The next generation", in *TUGboat*, Vol. 13, No. 2, pp. 138–138, July 1992.

[9] Philip Taylor: "The future of TEX", in *TUGboat*, Vol. 13, No. 4, pp. 433–442, December 1992.

[10] Philip Taylor: "NTS: the future of TEX", in *TUGboat*, Vol. 14, No. 3, pp. 177–182, October 1993.

[11] Philip Taylor: "NTS update", in *TUGboat*, Vol. 14, No. 4, pp. 381–382, December 1993.

[12] Philip Taylor: "Report of the 2nd meeting of the NTS group, February 1994", in *TUGboat*, Vol. 15, No. 2, pp. 96–97, June 1994.

[13] Philip Taylor: "Minutes of the NTS meeting held at Lindau on October 11/12th 1994", in *TUGboat*, Vol. 15, No. 4, pp. 434–437, December 1994.

[14] Philip Taylor: "NTS & ε-TEX: a status report", in *TUGboat*, Vol. 18, No. 1, pp. 6–12, March 1997.

[15] Zlatuška, Jiří (ed): *EuroTEX '92 Proceedings*, pp. 235-254, September 1992. Published by CSTUG, Czechoslovak TEX Users Group, ISBN 80-210-0480-0.