JAVA and TEX

Timothy Murphy
School of Mathematics
Trinity College Dublin
Ireland
tim@maths.tcd.ie

Abstract

TEX and METAFONT, translated into JAVA and compiled with TYA, a public-domain JIT compiler, run about 10 times more slowly than the same programs in C (without TYA, they are 20 times slower). A year ago, they ran 50 times more slowly. In a year's time, perhaps using Sun's new JIT compiler, it is reasonable to assume that the factor will be down to 2 or 3.

At that point — bearing in mind the speed with which the speed of computers is increasing — TEX-in-JAVA will be a perfectly plausible alternative to TEX-in-C; and then we shall have to weigh its lack of pace against the several advantages that JAVA has to offer, such as, portability, "netability", modularity, threads, and graphics.

Has TEX found its natural niche?

TEX has attained a complete monopoly of the mathematical market. (Are there still primitive people somewhere in the world speaking `eqn`?) And as mathematics continues its remorseless march to colonize new areas of knowledge, it carries TEX (like a disease) with it.

At the same time, it must be admitted that TEX has been less successful outside these areas than was hoped for, say 10 years ago. Of course that is not a disaster. According to Ken Thompson (creator of UNIX), "a program should do one thing, and do it well", and it may be that mathematical typesetting is the one thing that TEX does well, indeed superbly well. It would be foolish to risk this in pursuit of some universal rôle.

However, the cause is not necessarily lost. In the author's view, the solution does not lie in the addition of yet more features to TEX/LATEX — features which all too often satisfy the needs of the cognoscenti at the cost of complication for the newcomer — but rather in a more rigorous analysis of the TEX engine, and of the function and relation of its parts. It is the author's thesis that JAVA can provide the stimulus to set such an analysis in train.

It should be emphasised that the author is not suggesting — indeed, would be bitterly opposed to — the creation of yet another "near-TEX". The only threat that TEX faces in the medium term is fragmentation. All religions agree on one thing: that the greatest danger comes from within. Today, none of the schismatic versions of TEX (with the possible exception of PDFTEX, which denies the accusation of heresy) has a measurable share of the market; but if NTS, let us say, were to gain the allegiance of 25% of TEX users then the future of TEX — and NTS — would be in doubt. (For a more sympathetic view of NTS and other TEX extensions, see Knappen, 1995.)

The danger of such fragmentation can be seen clearly in the failure of literate programming' to fulfil the promise vested in it by Knuth (1992). The proliferation of innumerable *`WEB` programs (and one should include with these the LATEX `doc` system) — each of them doubtless superior in some aspect to Knuth's original — far from leading to widespread adoption of the literate programming paradigm, has stifled it almost to death.

The JAVATEX project

Although the principal aim of this talk is to demonstrate `DviPdf`, a TEX output driver written in JAVA, it may be more useful in this written version to say a little about the JAVATEX project of which `DviPdf` is part. This project has two main thrusts:

1. To translate the classic `.web` files (`tex.web`, `mf.web`, `tangle.web`, etc) into JAVA, using `web2java`, a straightforward modification of the standard `web2c` translator.

2. To develop output drivers — and other TEX support programs — in JAVA, using the standard Knuth/Levy `cweb`, modified (slightly) to output JAVA rather than C.

Why Java? JAVA has several advantages over C as a medium for TEX software.

Portability: In principle, a JAVA application — expressed as a number of communicating JAVA classes — should work unchanged on all platforms supporting JAVA, which means, in effect, under every OS.

Netability: JAVA was designed with the Internet in mind, and its adoption should allow TEX to be integrated more easily into the Web.

Modularity: JAVA is *object-oriented*, allowing classes to be shared by different programs, so that, for example, all drivers can share the same font manager and file server, and use the same DVI reader, And one can define an abstract generic driver, minimizing the size of actual drivers.

More speculatively, although TEX and METAFONT are large monolithic programs, they are actually written in a modular style — almost as though Knuth had JAVA in mind! — and it should be relatively simple to "hive off" font routines, for example, as a separate `TeXFont` class, without modifying the essential code in any way. Breaking up TEX (or METAFONT) in this way into a number of co-operating classes might mean that variations such as PDFTEX and METAPOST could be implemented as relatively small extensions of one or more of these classes.

Graphics: The standard graphical interface built into JAVA — but interpreted in the style of the platform in use — should mean that the same TEX viewer can function under UNIX, MS-DOS and Mac. And this interface would also offer a graphical alternative to the perhaps old-fashioned text-based interface traditional to TEX.

Threads: There are some advantages in running the different parts of a program as separate threads. For example, a font server can "sleep" until a font is requested; in an integrated system it may serve more than one program or even more than one user. By running TEX and friends as "threads", last-minute changes (as, for example, changing the sizes of arrays) can be implemented before the thread starts, and a program can pause while some intermediate task is performed, before resuming where it left off.

But JAVA is so slo … ow? But that is part of its charm!

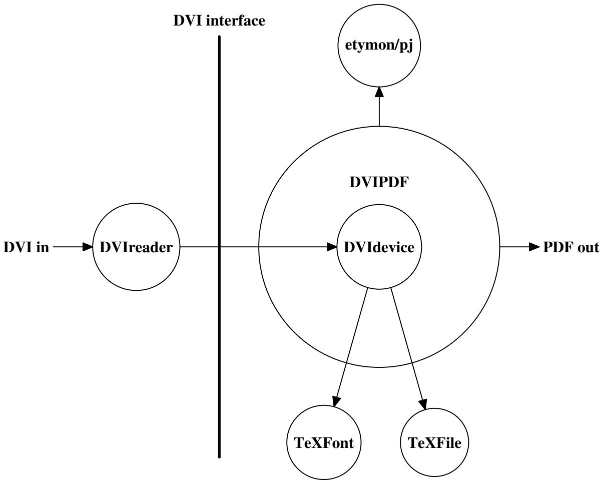

Figure 1: Anatomy of a driver

What is this world if, full of care
We have no time to stand and stare.

William Henry Davies (1870–1940)

At least things are getting better — three years ago, JAVA was 50 times as slow as C. Today, it is only five times as slow (with JIT compiler). Hopefully, this ratio will slowly approach a limiting value between two and three.

Sadly, Sun's long-awaited HotSpot compiler — now available (free of charge) on several platforms (Sun Microsystems, 1999) — failed signally to fulfil its promise that it would make JAVA applications run as fast as those in C++. It turns out to be little better than other JIT compilers.

A TEX output driver

Although we have chosen to illustrate our talk with DVIPDF — translating DVI input into PDF output — most of the code is common to all our TEX output drivers. The program is divided into seven parts (Figure 1):

The `DviReader`: This reads the DVI document, and translates the `DVI` commands into 'messages', as specified by

The `DVI` interface: This provides a "*cordon sanitaire*" between the `DviReader` and the driver proper.

The `DviDevice`: All output drivers share a great deal of functionality. For example, all treat fonts in much the same way. JAVA allows us to define a generic, or *abstract*, driver — `DviDevice` — containing this shared code. This abstract driver `implements DVI`; that is to say, it provides methods responding to the messages sent by the DviReader, as specified by the DVI interface.

The DviPdf driver: Several of the methods provided by the generic driver DviDevice are empty; it is left to each concrete driver — such as DviPdf — which "extends DviDevice" to provide proper methods in these cases.

TeXFont: From TEX's point of view, all fonts look much the same. We express this by defining an abstract TeXFont class. Each font type — .pk fonts, Type 1 or Type 3 PostScript fonts, .tfm font descriptors, virtual fonts, etc — extends this class by adding its own particularities.

TeXFile: This is our "file manager", a highly simplified analogue of kpathsea. In effect, it uses JAVA's Hashtable class to construct a database of the TEXMF tree (or trees).[1]

The PDF classes: PDF — Adobe's anointed successor to PostScript — is object oriented, and thus particularly well-suited to JAVA.

Fortunately, there is an excellent library of JAVA classes — the pj library from Etymon Systems (1999), freely available with source — for reading and writing PDF files. Each kind of PDF object — font, page, etc — is represented by a pj class, with methods appropriate to that object.

In effect, the job of DVIPDF is simply to build up a PDF object; it can then be left to the pj classes to present that object to the world.

The DVI interface. The JAVA *interface* provides an exemplary tool for dissecting an application (i.e. a program) into independent parts which communicate according to the strict protocol laid down in the interface definition.

The DVI interface specifies 15 kinds of "messages". Any driver that implements DVI must provide 15 methods for responding to these messages. Since the definition of the interface is short and sweet, we give it in its entirety:

```
public interface DVI {

  void moveRight( int dh );
  void moveDown( int dv );
  void moveTo( int h, int v );

  void defineFont( int f, int checkSum,
    int scaledSize, int designSize,
    String area, String name );
  boolean setFont( int f );
  int setChar( int c );
  void putChar( int c );
  void setRule( int wd, int ht );
  void putRule( int wd, int ht );

  void special( String message );

  void bop( int count[] );
  void eop();

  void preamble( int numerator,
    int denominator, int mag,
    String comment );
  void postamble( int tallestPage,
    int widestPage, int maxStackDepth,
    int noOfPages );

  DataInputStream dviStream( int c );
}
```

All these methods (except the last) will be more or less self-explanatory to those familiar with the DVI format.

The first two "motion methods", moveRight() and moveDown(), describe relative motion, while the third, moveTo(), is absolute.

We note that while communication is primarily *from* the DviReader *towards* the "front end" of the driver, information can be passed back through the return value of the function or method. Thus setChar() returns the width of the character (which is all the DviReader needs to know about it), while setFont() returns true or false according as the font is *virtual* or *real*.

The last method, dviStream(), is the only one which is not immediately suggested by the DVI format. It is required to implement virtual fonts. A virtual character — that is, a character in a virtual font — consists of a fragment of DVI code, which must be integrated into the DVI document proper. In effect the input stream must be temporarily diverted to the sequence of DVI commands constituting that character.

The DviReader knows if the current font is virtual, from the return value of the last setFont(). If that is so then every character encountered until the next setFont() is virtual. After reading such a character, say character number c, the DviReader sends the dviStream(c) message to the device, which consults the appropriate font and points the reader to the new stream. (We use here the nice property of JAVA, that it can treat information in a file, and in a string, on the same footing.)

[1] Am I alone in finding kpathsea excessively complex? The bureaucracy of TDS (the TEX Directory Structure) seems to me entirely misplaced. Surely the computer was designed precisely to relieve us of such tedious (pun intended) tasks? Does it really matter if .tfms and .pks and .stys find themselves in the same bed?

Virtual fonts and superfonts. A virtual font contains a number of *local* fonts. These are normally *real* fonts, but in principle they could themselves be virtual fonts.

This recursive potential of virtual fonts does not seem to have been exploited. It means in effect that the fonts in a TeX document form a tree, the leaves of which are real fonts, while the internal nodes are virtual fonts.

It is a natural step to connect the set of fonts by introducing a *superFont*, of which the top-level fonts — those actually named in the DVI document — are local fonts.

Recall that a virtual character is a fragment of DVI code. This suggests that we might regard the DVI document itself as a character — let us say character 0 — in the superFont.

It is amusing to take this conceit a little further. Different DVI documents could be characters 1, 2, 3, . . . , in the same superfont. Moreover, the superfont could itself be a local font in a super-superfont, which could itself . . . . A whole library of TeX documents might be organised in this way.

Tools

It is an essential feature of the JAVATeX project that all code in the package is written in Knuth/Levy `cweb` format, slightly modified (as described below) to output JAVA rather than C.

Thus the `DviPdf` driver is encoded in the files `DVI.w`, `DviDevice.w`, `TeXFile.w`, `TeXFont.w`, etc. (By convention, `cweb` source files carry the extension `.w`, to distinguish them from the classic Pascal-based `WEB` files, which carry the extension `.web`.)

As mentioned earlier, the JAVATeX project also encompasses the translation of Knuth's classic `WEB` programs into JAVA, using `web2java`, a development — in some ways, a simplification — of `web2c`, the core program in the UNIXTeX implementation of TeX and its relations.

As an exercise, we base our `DviReader` on `dvitype.web`, which Knuth provided as a model for drivers. Thus `DviReader` is defined by a change file `DviReader.ch` to `dvitype.web`. The resulting Pascal file `DviReader.p` is then translated into `DviReader.java` by `web2java`.

We end this note with a necessarily brief description of these basic JAVATeX tools.

Cweb for JAVA. Knuth's original `web` format was tied to Pascal. Later Knuth and Levy developed `cweb` to provide output in C. Since JAVA is a dialect of C, `cweb` only requires minor modifications to output JAVA. These are contained in the change files `ctang-java.ch`, `cweav-java.ch` and `comm-java.ch`. If `ctangle` and `cweave` are compiled with these change files (as, for example, by modifying the `cweb Makefile` by changing the line `TCHANGES=` to `TCHANGES=ctang-java.ch`, and similarly for `WCHANGES` and `CCHANGES`). then the `+j` switch[2] can be used to output JAVA:

```
% ctangle +j DVI.w
```

produces the file `DVI.java`, which can then be compiled in the usual way

```
% javac DVI.java
```

On the other hand, the documentation is produced by

```
% cweave +j DVI.w
```

creating the LaTeX file `DVI.tex`, which can then be processed in the usual way

```
% latex DVI
% xdvi DVI
% dvips DVI
```

Ctangle. In passing from `web` to `cweb`, Knuth and Levy dispensed with the macro feature `@d`, on the grounds that its functionality was more than adequately provided by C's `#define`.

However, JAVA in turn has dispensed with `#define`, so it seemed useful to transplant back this lost feature from `tangle` to `ctangle`. Fortunately, this turned out to be relatively simple since the amputation had been crude and the stumps remained. This allows us, for example, to say in `DVI.w` (and elsewhere)

```
@d DviUnit == int
```

and then

```
void moveRight( DviUnit dh );
```

This clarifies the code and also makes it simpler to change the type of `DviUnit` if that should prove desirable.

Cweave. The changes to `cweave`, although more trivial, proved surprisingly tricky. The problem is that `cweave` (like `weave`) is based on a table of "productions" — a kind of pseudo-syntax which allows scraps of code to be "reduced". It turned out that JAVA required some five new production rules to add to the 100 or so rules for C . . .

Web2java. `Web2java` — like `web2c` — is a post-processor to `tangle`. To create `foo.java` from `foo.web` and `foo.ch` one first runs `tangle`:

```
tangle foo.web foo.ch
```

[2] The use of `+` rather than `-` as a prefix for switches is a feature of `cweb`.

This creates the Pascal (or pseudo-Pascal) file `tangle.p` (or `tangle.pas` on some systems).[3] Class files are machine-independent — provided "native methods" are eschewed, and care taken to avoid such OS-specific idioms as '`\n`' for end-of-line, rather than JAVA's '`line.separator`'— so `tangle.class` from the JAVATeX distribution should run on any system. Note that this file, like all JAVATeX programs, is defined to be in the `javaTeX` package, and so must be placed in a subdirectory called `javaTeX` relative to the `CLASSPATH`. Note too that JAVA refers to this class as `javaTeX.tangle`, rather than `javaTeX/tangle`, as one might expect.

This file is then passed through `web2java` to create `foo.java`:

$$\texttt{foo.web}+\texttt{foo.ch}\xrightarrow{\texttt{tangle}}\texttt{foo.p}$$
$$\xrightarrow{\texttt{web2java}}\texttt{foo.java}.$$

Actually, this is a slight oversimplification. The file `common.defines` is prepended to `foo.p` *before* passing through `web2java`:

$$\texttt{common.defines}+\texttt{foo.p}\xrightarrow{\texttt{web2java}}\texttt{foo.java}.$$

All this is completely analagous to `web2c`, except that we are able to dispense with the post-processor `fixwrites`, for JAVA I/O contains nothing as exotic as C's `printf`.

The filter `web2java` is created by the programs `flex` and `bison` (or `lex` and `yacc`) from the files `web2java.l` and `web2java.y`. This is completely analagous to `web2c`. The `lex/flex` file `web2java.l` is the same as `web2c.l`, with the addition of a small number of new tokens: `new`, `try`, `catch`, `throw`, `throws`, etc. The syntax description in `web2java.y` has rather more changes, compared with `web2c.y`.

On the plus side, since JAVA has no pointers all the pointer-related material has been deleted. There is no attempt to determine if a function argument is "formal var" or not; and no need, therefore, to re-name functions with such arguments.

On the other hand, the introduction of class and object tokens necessarily adds to the number of rules in `web2java.y`. Thus variables and functions can be preceded by a `CLASSIFIER`, consisting of a (possibly empty) sequence of `class_id_tok`s and `object_id_tok`s followed by periods ('.'s). For those familiar with `web2c`, a short excerpt from `web2java.y` should give the idea:

```
CLASSIFIER:
  /* empty */
| CLASSIFIER class_id_tok '.'
  {
    my_output(last_id);
    my_output(".");
  }
| CLASSIFIER SIMPLE_OBJECT '.'
  {
    my_output(".");
  }
;

SIMPLE_OBJECT:
  object_id_tok
    {
      my_output (last_id);
    }
  VAR_DESIG_LIST
| object_id_tok
  {
    my_output (last_id);
  }
;
```

But for the most part translating `WEB` to JAVA is, if anything, simpler than `WEB` to C. One apparent difficulty is the lack of a pre-processor in JAVA, since `web2c` leaves a good deal of work to `cpp`. This means that more must be done in the change file, which is probably A Good Thing. The three main issues which arise are:

- the absence of `goto`s in JAVA
- the lack of `typedef`s in JAVA
- input/output

These are discussed briefly in the following three subsections.

Removing `goto`s. Java has no `goto`; in compensation, it allows `break` and `continue` statements to carry a *label*, as in `break lab21` or `continue lab3`, for example. The corresponding labels must appear at the beginning of the loop in question. (A `break` label can also be attached to a `switch` statement, but we make no use of that.) If a `break` or `continue` statement has no label, it is understood to refer to the smallest loop (or switch) enclosing the statement. Thus, labelling is only required in the case of nested loops or switches.

Fortunately, Knuth has followed a strict protocol in the classic `WEB` files. Raw `goto`s (as in `goto 40`) very rarely appear. In almost all cases a label is used, as in `goto found`, where `found` has earlier been defined as

```
@d found=40
```

In effect, the `goto`s are divided into a small number of cases, according to their function. By far the most frequent of these cases are: `goto break` to break out

[3] If you like driving in the slow lane, you could run the JAVA `tangle` instead: `java javaTeX.tangle foo.web foo.ch`.

of a loop, goto continue to continue around a loop, and return to return from a routine.

This protocol allows most gotos to be processed automatically. Thus goto break is translated as break and goto continue as continue, while return is translated as return (with the appropriate value in the case of a function).

However, in perhaps a third of the gotos, labels must be inserted by hand, for example, as a break out of an outer loop. Note that in such a case the label in the WEB file is almost certainly in the wrong place, for, by Knuth's convention, break means "break to the end of the loop" while JAVA requires the label to appear at the start of the loop. web2java takes advantage of this by deleting the label from a break or continue statement unless that label has already appeared (in the current routine) before the statement.

Of course a goto may not go to the beginning or end of a loop; in that case a new "artificial" loop must be inserted, with a break at the end to ensure that it is only traversed once.

All this is rather messy and could probably be automated to a much greater extent. At least some checks have been introduced in web2java.y, to verify as far as possible that the new code has the same effect as the old.

Type definitions. There are no typedefs in Java. In theory one could replace typedefs by class definitions but that would add considerable complication to the code. Instead, we simply change them to substitutions (as though in C changing typedefs to #define's). So, for example, we make the change

```
@x
@<Types...@>=
@!ASCII_code=0..255;
@y
@d ASCII_code==0..255
@z
```

Later web2java will replace this range 0..255 by an appropriate type (currently int). This entails some changes in web2java.y, to allow ranges for procedure and function parameters such as:

```
procedure p(x:0..255);
```

Presently all ranges are replaced by int, since Java is rather strict about type conversion, and requires casting where C does not.

Input/Output. JAVA I/O is much closer to PASCAL syntax than is C. Thus

```
write_ln(term_out, ’value is ’, v);
```

in PASCAL becomes

```
System.out.println("value is " + v);
```

in JAVA. This allows us to incorporate I/O into web2java.y, dispensing with the fixwrites postprocessor required by web2c.

The only unusual feature of JAVA I/O is that most I/O statements must be contained in a try statement, which in turn must be followed by a catch statement to catch any I/O 'errors'. However, this is perfectly straightforward, as may be illustrated by an I/O function from DviReader.ch:

```
function signed_pair:integer;
  {returns the next two bytes, signed}
var a,@!b:eight_bits;
begin a:=0; b:=0;
try begin a:=dvi_file.readByte;
  b:=dvi_file.readUnsignedByte; end;
catch (ex: IOException) begin
  EOF_dvi_file:=true; end;
if EOF_dvi_file then signed_pair:=0
else begin cur_loc:=cur_loc+2;
  signed_pair:=a*256+b; end;
end;
```

Conclusion

Hopefully, this all-too-brief tour has given some taste of the JAVATeX project. All comments, contributions and suggestions will be gratefully received.

The project (and all its parts) is freely available (Murphy, 1999). For simplicity it is published subject to the GNU GPL Licence, Essentially this allows the work to be freely copied and used, provided the original files DVI.w, etc, are made available. Changes should preferably be made through change files, e.g., DVI.ch.

References

Etymon Systems. "Java software for parsing, manipulating, and creating Adobe PDF file". http://www.etymon.com/pj/, 1999.

Knappen, Börg. "NTS-FAQ". CTAN/info/NTS-FAQ, 1995. (In these references, CTAN denotes any of the CTAN sites, eg ftp://ftp.tex.ac.uk/pub/tex or ftp://ftp.dante.de/pub/tex).

Knuth, Donald E. *Literate Programming*. CSLI, Stanford, 1992.

Murphy, Timothy. "The JAVATeX project". http://www.maths.tcd.ie/~tim/javaTeX, 1999. Also available from CTAN/systems/java/javatex.

Sun Microsystems. "Java HotSpot Performance Engine". http://java.sun.com/products/hotspot/, 1999.