MKII–MKIV

Hans Hagen
Hasselt, The Netherlands
`http://luatex.org`
`http://pragma-ade.com`
`pragma (at) wxs dot nl`

Abstract

Some time ago the pdfTEX development team set a road map for the next generation of pdfTEX. It was decided that within a reasonable timeframe pdfTEX would go 24/32 bit and support OpenType fonts. At the same time, after some preliminary experiments, it was decided that it made sense to embed the Lua scripting engine into TEX.

Currently, Taco Hoekwater and I spend a lot of time exploring the possibilities of a TEX engine enhanced in this way. Downward compatibility as well as the traditional stability are very important conditions in this proces. Because we're both actively involved in the development of ConTEXt and both know the internals quite well, we can easily test Lua based alternatives and explore possible routes and needed extensions.

This paper presents some possible features of LuaTEX (some may go, some may change, others will be added). I will discuss how they may influence the way ConTEXt will be organized in the future as well as how code development may change in nature and influence users. I will also demonstrate some of the features of that LuaTEX currently offers.

1 Introduction

The development of LuaTEX started with a few email exchanges between me and Hartmut Henkel. I had played a bit with Lua in SCITE and somehow felt that it would fit into TEX quite well. Hartmut made me a version of pdfTEX which provided a `\lua` command. After exploring this road a bit Taco Hoekwater took over and we quickly reached a point where the pdfTEX development team could agree on following this road to the future.

The development was boosted by a substantial grant from Colorado State University in the context of Professor Idris Samawi Hamid's Oriental TEX Project. This project aims at bringing features into TEX that will permits ConTEXt to do high quality Arab typesetting. Due to this grant Taco could spend substantial time on development, which in turn meant that I could start playing with more advanced features.

This document is not so much a manual as a report on the state of affairs. Things may evolve and the way things are done may change, but it felt right to keep track of the process.

2 From Mark II to Mark IV

In 2005 the development of LuaTEX started, a further development of pdfTEX and a precursor to pdfTEX version 2. This TEX variant will provide:

- 21–32 bit internals plus a code cleanup
- flexible support for OpenType fonts
- an internal UTF data flow
- the bidirectional typesetting of Aleph
- Lua callbacks to relevant TEX internals
- some extensions to TEX (for instance math)
- efficient communication with MetaPost

In the tradition of TEX this successor will be downward compatible in essential ways and in the end, there is still pdfTEX version 1 as fall back.

In the mean time we have seen another Unicode variant show up: XƎTEX, which is under active development, uses external libraries, provides access to the fonts on the operating system, etc.

From the beginning, ConTEXt always worked with all engines. This was achieved by conditional code blocks: depending on what engine was used, different code was put in the format and/or used at runtime. Users normally were unaware of this. Examples of engines are ε-TEX, Aleph, and XƎTEX. Because nowadays all engines provide the ε-TEX features, in August 2006 we decided to consider those features to be present and drop providing the standard TEX compatible variants. This is a small effort because all code that is sensitive for optimization already has ε-TEX code branches for many years.

However, with the arrival of LuaTEX, we need a more drastic approach. Quite a lot existing code can go away, to be replaced by different solutions. Where TEX code ends up in the format file, along with its state, Lua code will be initiated at run time, after a Lua instance is started. ConTEXt reserves its own instance of Lua.

Most of this will go unnoticed for the users because the user interface will not change. For developers however, we need to provide a mechanism to deal with these issues. This is why, for the first time in ConTEXt's history, we will officially use a kind of version tag. When we changed the low level interface from Dutch to English we jokingly talked of version 2. So, it makes sense to follow this lead.

- ConTEXt Mark I At that moment we still had a low level Dutch interface, invisible for users but not for developers.
- ConTEXt Mark II We now have a low level English interface, which we hoped (and indeed saw happen) would trigger more development by users.
- ConTEXt Mark IV This is the next generation of ConTEXt, with parts re-implemented. At some points, it's a drastic system overhaul.

Keep in mind that the functionality does not change, although in some places, for instance fonts, Mark IV may provide additional functionality. Most users will not notice the difference (maybe apart from performance and convenience) since at the user interface level nothing changes (most of it deals with typesetting, not low level details).

The hole in the numbering permits us to provide a Mark III version as well. Once XƎTEX is stable, we may use that slot for XƎTEX specific implementations.

As of August 2006 the banner has been adapted to this distinction:

```
ver: 2006.09.06 22:46 MK II  fmt: 2006.9.6
ver: 2006.09.06 22:47 MK IV  fmt: 2006.9.6
```

This numbering system is reflected at the file level in such a way that we can keep developing the way we do, i.e. no files all over the place, in subdirectories, etc.

Most of the system's core files are not affected, but some may be, like those dealing with fonts, input and output encodings, file handling, etc. Those files may come with different suffixes:

- `somefile.tex`: the main file, implementing the interface and common code
- `somefile.mkii`: mostly existing code, suitable for good old TEX (ε-TEX, pdfTEX, Aleph).
- `somefile.mkiv`: code optimized for use with LuaTEX, which could follow completely different approaches
- `somefile.lua`: Lua code, loaded at format generation time and/or runtime

As said, some day `somefile.mkiii` code may show up. Which variant is loaded is determined automatically at format generation time and/or at run time.

3 How Lua fits in

Introduction

Here I will discuss a few of the experiments that drove the development of LuaTEX. It describes the state of affairs around the time that we were preparing for TUG 2006. This development was rather demanding for Taco and me but also much fun. We were in a kind of permanent Skype chat session, with binaries flowing in one direction and TEX and Lua code the other way. By gradually replacing (even critical) components of ConTEXt we had a real test bed and torture tests helped us to explore and debug at the same time. Because Taco uses Linux as platform and I mostly use Windows, we could investigate platform dependent issues conveniently. While reading this text, keep in mind that this is just the beginning of the game.

I will not provide sample code here. When possible, the Mark IV code transparently replaces Mark II code and users will seldom notices that something happens in different way. Of course the potential is there and future extensions may be unique to Mark IV.

Compatibility

The first experiments, already conducted with the experimental versions involved runtime conversion of one type of input into another. An example of this is the (TI) calculator math input handler that converts a rather natural math sequence into TEX and feeds that back into TEX. This mechanism eventually will evolve into a configurable math input handler. Such applications are unique to Mark IV code and will not be backported to Mark II. The question is where downward compatibility will become a problem. We don't expect many problems, apart from occasional bugs that result from splitting the code base, mostly because new features will not affect older functionality. Because we have to reorganize the code base a bit, we also use this opportunity to start making a variant of ConTEXt which consists of building blocks: MetaTEX. This is less interesting for the average user, but may be of interest for those using ConTEXt in workflows where only part of the functionality is needed.

MetaPost

Of course, when I experiment with such new things, I cannot let MetaPost leave untouched. And so, in this early stage of LuaTEX development I decided to play with two MetaPost related features: conversion and runtime processing.

Conversion from MetaPost output to PDF is currently done in pure TEX code. Apart from convenience, this has the advantage that we can let TEX take care of font inclusions. The tricky part of this conversion is that MetaPost output has some weird aspects, like dvips specific linewidth snapping. Another nasty element in the conversion is that we need to transform paths when pens are used. Anyhow, the converter has reached a rather stable state by now.

One of the ideas with MetaPost version 1^+ is that we will have an alternative output mode. From the perspective of LuaTEX it makes sense to have a Lua output mode. Whatever converter we use, it needs to deal with Metafun specials. These are responsible for special features like transparency, graphic inclusion, shading, and more. Currently we misuse colors to signal such features, but the new pre/post path hooks permit more advanced implementations. Experimenting with such new features is easier in Lua than in TEX.

The Mark IV converter is a multi-pass converter. First we clean up the MetaPost output, then convert the PostScript code into Lua calls. We assume that this Lua code can eventually be output directly from MetaPost. We then evaluate this converted Lua blob, resulting in TEX commands. Example:

```
1.2 setlinejoin
```

turned into:

```
mp.setlinejoin(1.2)
```

becoming:

```
\PDFcode{1.2 j}
```

which is, when the pdfTEX driver is active, equivalent to:

```
\pdfliteral{1.2 j}
```

Of course, when paths are involved, more things happen behind the scenes, but in the end an `mp.path` enters the Lua machinery.

When the Mark IV converter reached a stable state, tests demonstrated then the code was up to 20% slower that the pure TEX alternative on average graphics, and but faster when many complex path transformations (due to penshapes) need to be done. This slowdown was due to the cleanup (using expressions) and intermediate conversion. Because Taco both develops LuaTEX and maintains and extends MetaPost, we conducted experiments that combine features of these programs. As a result of this, shortcuts found their way into the MetaPost output.

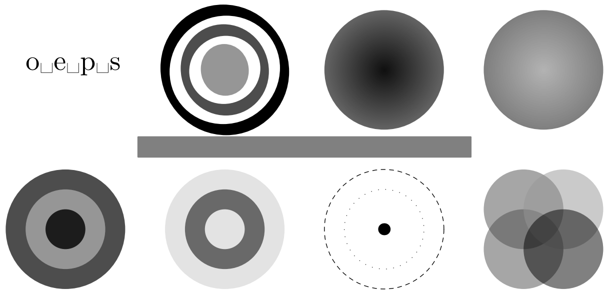

Figure 1 Our converter test figure.

Cleaning up the MetaPost output using Lua expressions takes relatively much time. However, starting with version 0.970 MetaPost uses a preamble, which permits not only short commands, but also gets rid of the weird linewidth and filldraw related PostScript constructs. The moderately complex graphic that we use for testing (figure 1) takes over 16 seconds when converted 250 times. When we enable shortcuts we can avoid part of the cleanup and runtime goes down to under 7.5 seconds. This is significantly faster than the Mark II code. We did experiments with simulated Lua output from MetaPost and then the Mark IV converter really flies. The values on Taco's system are given in parentheses:

prologues/ mpprocset	**1/0**	**1/1**	**2/02/1**
Mark II	8.5 (5.7)	8.0 (5.5)	8.8 8.5
Mark IV	16.1 (10.6)	7.2 (4.5)	16.3 7.4

The main reason for the huge difference in the Mark IV times is that we do a rigorous cleanup of the older MetaPost output in order to avoid the messy (but fast) code that we use in the Mark II converter. Think of:

```
0 0.5 dtransform truncate idtransform
  setlinewidth pop
closepath gsave fill grestore stroke
```

In the Mark II converter, we push every number or keyword on a stack and use keywords as trigger points. In the Mark IV code we convert the stack based PostScript calls to Lua function calls. The top-level `0...pop` and `closepath...stroke` expressions are each converted to single calls first. When `prologues` is set to 2, such lines no longer show up and are replaced by simple calls accompanied by definitions in the preamble. Not only that, instead of verbose keywords, one or two character shortcuts are used. This means that the Mark II code can be faster when procsets are used because shorter strings end up in the stack and comparison happens faster. On the other hand, when no procsets are used, the runtime is longer because of the larger preamble.

Because the converter is used outside ConTeXt as well, we support all combinations in order not to get error messages, but the converter is supposed to work with the following settings:

```
prologues := 1 ;
mpprocset := 1 ;
```

We don't need to set `prologues` to 2 (font encodings in file) or 3 (also font resources in file). So, in the end, the comparison in speed comes down to 8.0 seconds for Mark II code and 7.2 seconds for the Mark IV code when using the latest greatest MetaPost. When we simulate Lua output from MetaPost, we end up with 4.2 seconds runtime and when MetaPost could produce the converter's TeX commands, we need only 0.3 seconds for embedding the 250 instances. This includes TeX taking care of handling the specials, some of which demand building moderately complex PDF data structures.

But, conversion is not the only factor in convenient MetaPost usage. First of all, runtime MetaPost processing takes time. The actual time spent on handling embedded MetaPost graphics is also dependent on the speed of starting up MetaPost, which in turn depends on the size of the TeX trees used: the bigger these are, the more time KPSE spends on loading the `ls-R` databases. Eventually this bottleneck may go away when we have MetaPost as a library. (In ConTeXt one can also run MetaPost between runs. Which method is faster depends on the amount and complexity of the graphics.)

Another factor in dealing with MetaPost, is the usage of text in a graphic (`btex`, `textext`, etc.). Taco Hoekwater, Fabrice Popineau and I did some experiments with a persistent MetaPost session in the background in order to simulate a library. The results look very promising: the overhead of embedded MetaPost graphics goes to nearly zero, especially when we also let the parent TeX job handle the typesetting of texts. A side effect of these experiments was a new mechanism in ConTeXt (and Metafun) where TeX did all typesetting of labels, and MetaPost only worked with an abstract representation of the result. This way we can completely avoid nested TeX runs (the ones triggered by MetaPost). This also works ok in Mark II mode.

Using a persistent MetaPost run and piping data into it is not the final solution if only because the terminal log becomes messed up too much, and also because intercepting errors becomes very messy. In the end we need a proper library approach, but the experiments demonstrated that we needed to go this way: handling hundreds of complex graphics that hold typeset paragraphs (being slanted and rotated and more by MetaPost), took mere seconds compared to minutes when using independent MetaPost runs for each job.

Characters

Because LuaTeX is UTF based, we need a different way to deal with input encoding. For this purpose there are callbacks that intercept the input and convert it as needed. For ConTeXt this means that the regime-related modules get Lua-based counterparts. As a prelude to advanced character manipulations, we already load extensive Unicode and conversion tables, with the benefit of being able to handle case handling with Lua.

The character tables are derived from Unicode tables and Mark II ConTeXt data files, and generated using mtxtools. The main character table is pretty large, and this made us experiment a bit with efficiency. It was in this stage that we realized that it made sense to use precompiled Lua code (using `luac`). During format generation we let ConTEXt keep track of used Lua files and compile them on the fly. For a production run, the compiled files were loaded instead.

Because at that stage LuaTEX was already a merge between pdfTEX and Aleph, we had to deal with pretty large format files. Thus, on 2006-09-18 the ConTEXt format with the English user interface amounted to:

luatex	**pdftex**	**xetex**	**aleph**
9 552 042	7 068 643	8 374 996	7 942 044

One reason for the large size of the format file is that the memory footprint of a 32-bit TEX is larger than that of good old TEX, even with some of the clever memory allocation techniques used in LuaTEX. After some experiments where size and speed were measured Taco decided to compress the format using level 3 zip compression. This brilliant move lead to the following sizes on 2006-10-23:

luatex	**pdftex**	**xetex**	**aleph**
3 135 568	7 095 775	8 405 764	7 973 940

The first zipped versions were smaller (around 2.3 meg), but in the meantime we moved the Lua code into the format and the character related tables take some space.

Debugging

In the process of experimenting with callbacks I played a bit with handling TEX error information. An option is to generate an HTML page instead of the usual blob of text on the terminal.

Playing with such features gives us an impression of what kind of access we need to TEX's internals. It also formed a starting point for conversion routines and a mechanism for embedding Lua code in HTML pages generated by ConTEXt.

File I/O

Replacing TEX's input and output handling is non-trivial. Not only is the code quite interwoven in the Web2C source, but there is also the KPSE library to deal with. This means that quite a few callbacks are needed to handle the different types of files. Also, there is output to the log and terminal to deal with.

Getting this done took us quite some time and testing and debugging was good for some headaches. The mechanisms changed a few times, and TEX and Lua code was thrown away as soon as better solutions came around. Because we were testing on real documents, using a fully loaded ConTEXt we could converge to a stable version after a while.

Getting this I/O stuff done is tightly related to generating the format and starting up LuaTEX. If you want to overload the file searching and I/O handling, you need overload as soon as possible. Because LuaTEX is also supposed to work with the existing KPSE library, we still have that as fallback, but in principle one could think of a KPSE free version, in which case the default file searching is limited to the local path and memory initialization also reverts to the hard coded defaults. A complication is that the soure code has KPSE calls and references to KPSE variables all over the place, so occasionally we run into interesting bugs.

Anyhow, while Taco hacked his way around the code, I converted my existing Ruby based KPSE variant into Lua and started working from that point. The advantage of having our own I/O handler is that we can go beyond KPSE. For instance, since LuaTEX has, among a few others, the zip libraries linked in, we can read from zip files, and keep all TEX related files in TDS compliant zip files as well. This means that one can say:

```
\input zip::somezipfile::somefile.tex
\input zip://some.zip/subdir/somefile.tex
```

and use similar references to access files. Of course we had to make sure that KPSE like searching in the TDS (standardized TEX trees) works smoothly. There are plans to link the curl library into LuaTEX, so that we can go beyond this and access network repositories.

Of course, in order to be more or less KPSE and Web2C compliant, we also need to support this paranoid file handling, so we provide mechanisms for that as well. In addition, we provide ways to create sandboxes for system calls.

Getting to intercept all log output (well, most log output) was a problem in itself. For this I used a (preliminary) XML based log format, which will make log parsing easier. Because we have full control over file searching, opening and closing, we can also provide more information about what files are loaded. For instance we can now easily trace what TFM files TEX reads.

Implementing additional methods for locating and opening files is not that complex because the library that ships with ConTEXt is already prepared for this. For instance, implementing support for:

```
\input http://example.net/somefile.tex
```

involved just a few lines of code, most of which deals with caching the files. Because we overload the whole I/O handling, this means that the following works ok:

```
\placefigure
  {http handling}
  {\externalfigure
    [http://www.pragma-ade.com/show-gra.pdf]
    [page=1,width=\textwidth]}
```

Other protocols, like FTP, are also supported, so one can say:

```
\typefile {ftp://anonymous:@ctan.org/\
  tex-archive/systems/knuth/lib/plain.tex}
```

On the agenda is playing with databases, but by the time that we enter that stage linking the `curl` libraries into LuaTEX should have taken place.

Verbatim

The advance of LuaTEX also permitted us to play with a long standing wish for catcode tables, a mechanism to quickly switch between different ways of treating input characters. An example of a place where such changes take place is verbatim (and, in ConTEXt, when dealing with XML input).

We had already encountered the phenomena that when piping back results from Lua to TEX, we needed to take care of catcodes so that TEX would see the input as we wished. Earlier experiments with applying `\scantokens` to a result and thereby interpreting the result conforming the current catcode regime was not sufficient or at least not handy enough, especially in the perspective of fully expandable Lua results. To be honest, `\scantokens` was rather useless for this purposes due to its pseudo file nature and its end-of-file handling but in LuaTEX we now have a convenient `\scantextokens` which has no side effects.

Once catcode tables were in place, and the relevant ConTEXt code adapted, I could start playing with one of the trickier parts of TEX programming: typesetting TEX using TEX, or verbatim. Because in ConTEXt verbatim is also related to buffering and pretty printing, all these mechanism were handled at once. It proved to be a pretty good test case for writing Lua results back to TEX, because anything you can imagine can and will interfere (line endings, catcode changes, looking ahead for arguments, etc). This is one of the areas where Mark IV code will make things look more clean and understandable, especially because we could move all kind of post-processing (needed for pretty printing, i.e. syntax highlighting) to Lua.

Pretty printing 1000 small (one line) buffers and 5000 simple `\type` commands perform as follows:

	TEX normal	TEX pretty	Lua normal	Lua pretty
buffer	2.5 (2.35)	4.5 (3.05)	2.2 (1.8)	2.5 (2.0)
inline	7.7 (4.90)	11.5 (7.25)	9.1 (6.3)	10.9 (7.5)

Between braces the runtime on Taco's more modern machine is shown. It's not that easy to draw conclusions from this because TEX uses files for buffers and with Lua we store buffers in memory. For inline verbatim, Lua calls bring some overhead, but with more complex content, this becomes less noticeable. Also, the Lua code is probably less optimized than the TEX code, and we don't know yet what benefits a Just In Time Lua compiler will bring.

XML

One interesting result is that the first experiments with XML processing don't show the expected gain in speed. This is due to the fact that the ConTEXt XML parser is highly optimized. However, if we want to load a whole XML file, for instance the formal ConTEXt interface specification `cont-en.xml`, then we can bring down loading time (as well as TEX memory usage) down from multiple seconds to a blink of an eye. Experiments with internal mappings and manipulations demonstrated that we may not so much need an alternative for the current parser, but can add additional, special purpose ones.

We may consider linking XSLTPROC into LuaTEX, but this is yet undecided. After all, the problem of typesetting does not really change, so we may as well keep the process of manipulating and typesetting separated.

Multipass data

Those who know ConTEXt a bit will know that it may need multiple passes to typeset a document. ConTEXt not only keeps track of index entries, list entries, cross references, but also optimizes some of the output based on information gathered in previous passes. Especially so-called “two-pass data” and positional information put some demands on memory and runtime. Two-pass data is collapsed in lists because otherwise we would run out of memory (at least this was true years ago when these mechanisms were introduced). Positional information is stored in hashes and has always put a bit of a burden on the size of a so-called utility file (ConTEXt stores all information in one auxiliary file).

These two datatypes were the first we moved to a Lua auxiliary file and eventually all information will move there. The advantage is that we can use efficient hashes (without limitations) and only need to run over the file once. And Lua is incredibly fast in loading the tables where we keep track of these things. For instance, a test file storing and reading 10 000 complex positions takes 3.2 seconds runtime with LuaTEX but 8.7 seconds with traditional pdfTEX. Imagine what this will save when dealing with huge files (400 page 300 Meg files) that need three or more passes to be typeset. And, now we can without problems bump position tracking to the millionth decimal place.

4 Initialization revised

Initializing LuaTEX in such a way that it does what you want it to do your way can be tricky. This has to do with the fact that if we want to overload certain features (using callbacks) we need to do that before the originals start doing their work. For instance, if we want to install our own file handling, we must make sure that the built-in file searching does not get initialized. This is particularly important when the built in search engine is based on the KPSE library. In that case the first serious file access will result in loading the `ls-R` filename databases, which will take an amount of time more or less linear with the size of the TEX trees. Among the reasons why we want to replace KPSE are the facts that we want to access zip files, do more specific file searches, use HTTP, FTP and whatever comes around, integrate ConTEXt specific methods, etc.

Although modern operating systems will cache files in memory, creating the internal data structures (hashes) from the rather dumb files take some time. On the machine where I was developing the first experimental LuaTEX code, we’re talking about 0.3 seconds for pdfTEX. One would expect a Lua based alternative to be slower, but it is not. This may be due to the different implementation, but for sure the more efficient file cache plays a role as well. So, by completely disabling KPSE, we can have more advanced I/O related features (like reading from zip files) at about the same speed (or even faster). In due time we will also support progname (and format) specific caches, which speeds up loading. In case one wonders why we bother about a mere few hundreds of milliseconds: imagine frequent runs from an editor or sub-runs during a job. In such situation every speed up matters.

So, back to initialization: how do we initialize LuaTEX. The method described here is developed for ConTEXt but is not limited to this macro package; when one tells TEXexec to generate formats using the `--luatex` directive, it will generate the ConTEXt formats as well as mptopdf using this engine.

For practical reasons, the Lua based I/O handler is KPSE compliant. This means that the normal `texmf.cnf` and `ls-R` files can be used. However, their content is converted in a more Lua friendly way. Although this can be done at runtime, it makes more sense to do this in advance using the luatools utility. The files involved are:

input	**raw input**	**runtime input / fallback**
	`ls-R`	`files.luc` / `files.lua`
`texmf.lua`	`temxf.cnf`	`configuration.luc` / `configuration.lua`

In due time luatools will generate the directory listing itself (for this some extra libraries need to be linked in). The configuration file(s) eventually will move to a Lua table format, and when a `texmf.lua` file is present, that one will be used.

```
luatools --generate
```

This command will generate the relevant databases. Optionally you can provide `--minimize` which will generate a leaner database, which in turn will bring down loading time to (on my machine) about 0.1 sec instead of 0.2 seconds. The `--sort` option will give nicer intermediate (`.lua`) files that are more handy for debugging.

When done, you can use luatools roughly like kpsewhich, for instance to locate files:

```
luatools texnansi-lmr10.tfm
luatools --all tufte.tex
```

You can also inspect its internal state, for instance:

```
luatools --variables  --pattern=TEXMF
luatools --expansions --pattern=context
```

This will show you the (expanded) variables from the configuration files. Normally you don't need to go that deep into the belly.

The luatools script can also generate a format and run LuaTEX. For ConTEXt this is normally done with the TEXexec wrapper, for instance:

```
texexec --make --all --luatex
```

When dealing with this process we need to keep several things in mind:

- LuaTEX needs a Lua startup file in both ini and runtime mode
- these files may be the same but may also be different
- here we use the same files but a compiled one in runtime mode
- we cannot yet use a file location mechanism

A `.luc` file is a precompiled Lua chunk. In order to guard consistency between Lua code and tex code, ConTEXt will preload all Lua code and store them in the bytecode table provided by LuaTEX. How this is done is another story. Contrary to these tables, the initialization code can not be put into the format, if only because at that stage we still need to set up memory and other parameters.

In our case, especially because we want to overload the I/O handler, we want to store the startup file in the same path as the format file. This means that scripts that deal with format generation also need to take care of (relocating) the startup file. Normally we will use TEXexec but we can also use luatools.

Say that we want to make a plain format. We can call luatools as follows:

```
luatools --ini plain
```

This will give us (in the current path):

```
120,808 plain.fmt
  2,650 plain.log
 80,767 plain.lua
 64,807 plain.luc
```

From now on, only the `plain.fmt` and `plain.luc` file are important. Processing a file

```
test \end
```

can be done with:

```
luatools --fmt=./plain.fmt test
```

This returns:

```
This is luaTeX, Version 3.141592-0.1-alpha-
20061018 (Web2C 7.5.5)
(./test.tex [1] )
Output written on test.dvi (1 page, 260 bytes).
Transcript written on test.log.
```

which looks rather familiar. Keep in mind that at this stage we still run good old Plain TEX. In due time we will provide a few files that will making work with Lua more convenient in Plain TEX, but at this moment you can already use, for instance, `\directlua`.

In case you wonder how this is related to ConTEXt, well only to the extent that it uses a couple of rather generic ConTEXt related Lua files.

ConTEXt users can best use TEXexec which will relocate the format related files to the regular engine path. In luatools terms we have two choices:

```
luatools --ini cont-en
luatools --ini --compile cont-en
```

The first case uses `context.lua` as the startup file. This Lua file creates the `cont-en.luc` runtime file. In the second case, luatools will create a `cont-en.lua` file and compile that one. An even more specific call would be:

```
luatools --ini --compile \
  --luafile=foo.lua cont-en
luatools --ini --compile \
  --lualibs=foo1.lua,foo2.lua cont-en
```

This call does not make much sense for ConTEXt. Keep in mind that luatools does not set up user specific configurations, for instance the `--all` switch in TEXexec will set up all patterns.

I know that it sounds a bit messy, but till we have a more clear picture of where LuaTEX is heading this is the way to proceed. The average ConTEXt user won't notice those details, because TEXexec will take care of things.

Currently we follow the TDS and Web2C conventions, but in the future we may follow different or additional approaches. This may as well be driven by more complex I/O models. For the moment extensions still fit in. For instance, in order to support access to remote resources and related caching, we have added to the configuration file the variable:

```
TEXMFCACHE = $TMP;$TEMP;$TMPDIR;$HOME;\
             $TEXMFVAR;$VARTEXMF;.
```

5 An example: CalcMath

introduction

For a long time TeX's way of coding math has dominated the typesetting world. However, this kind of coding is not that well suited for non-academics, such as schoolchildren. Often kids do know how to key in math because they use advanced calculators. So, when a couple of years ago we were implementing a workflow where kids could fill in their math workbooks (with exercises) on-line, it made sense to support so-called "Texas Instruments" math input. Because we had to parse the form data anyway, we could use `[[` and `]]` as math delimiters instead of `$`. The conversion took place right after the form was received by the web server.

By combining Lua with TeX, we can do the conversion from calculator math to TeX immediately, without auxiliary programs or complex parsing using TeX macros.

TeX

In a ConTeXt source one can use the `\calcmath` command, as in:

```
The strange formula
\calcmath{sqrt(sin^2(x)+cos^2(x))}
boils down to ...
```

One needs to load the module first, using:

```
\usemodule[calcmath]
```

Because the amount of code involved is rather small, eventually we may decide to add this support to the Mark IV kernel.

XML

Coding math in TeX is rather efficient. In XML one needs way more code. Presentation MathML provides a few basic constructs and boils down to combining those building blocks. Content MathML is better, especially from the perspective of applications that need to interpret the formulas. It permits for instance the ConTeXt content MathML handler to adapt the rendering to cultural driven needs. The OpenMath way of coding is like content MathML, but more verbose with fewer tags. Calculator math is more restrictive than TeX math and less verbose than any of the XML variants. It looks like this:

```
<icm>sqrt(sin^2(x)+cos^2(x))</icm> test
```

And in display mode:

```
<dcm>sqrt(sin^2(x)+cos^2(x))</dcm> test
```

Speed

This script (which you can find in the ConTeXt distribution as soon as the Mark IV code variants are added) is the first real TeX related Lua code that I've written; before this I had only written some wrapping and spell checking code for the SCITE editor. It also made a nice demo for a couple of talks that I held at usergroup meetings. The script has a lot of expressions. These convert one string into another. They are less powerful than regular expressions, but pretty fast and adequate. The feature I miss most is alternation like `(l|st)uck` but it's a small price to pay. As the Lua manual explains: adding a POSIX compliant regexp parser would take more lines of code than Lua currently does.

On my machine, running this first version took 3.5 seconds for typesetting 2500 times the previously shown square root of sine and cosine. Of this, 2.1 seconds were spent on typesetting and 1.4 seconds on converting. After optimizing the code, 0.8 seconds were used for conversion. A stand alone Lua takes .65 seconds, which includes loading the interpreter. On a test of 25 000 sample conversions, we could gain some 20% conversion time using the LuaJIT just in time compiler.