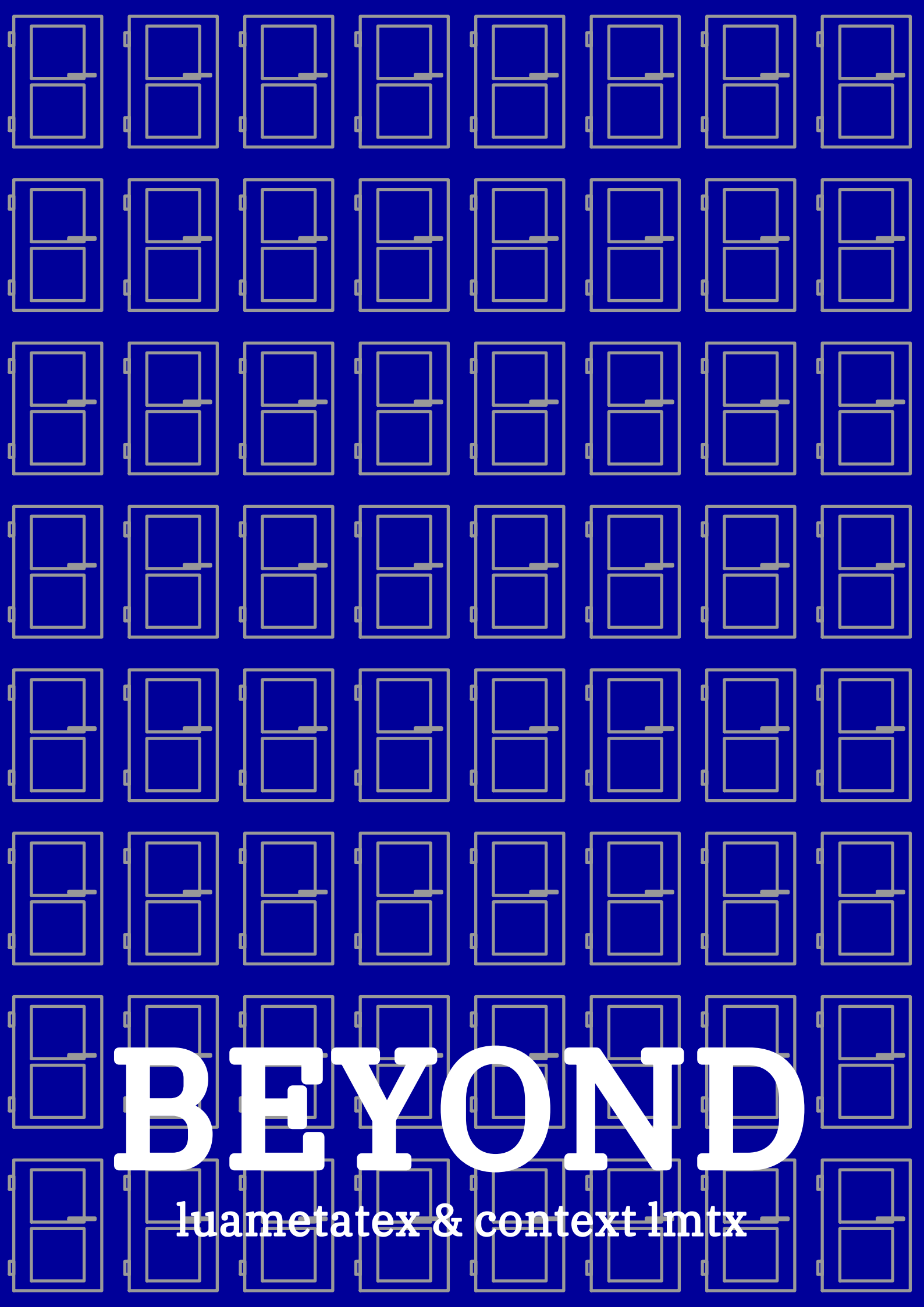

BEYOND

luametatex & context lmtx

Table of contents

1 Introduction 4

2 A new take on paragraphs 6

3 Twins 8

4 Namespaces 10

1 Introduction

This is the eights wrapup of the LuaTEX and LuaMetaTEX development cycle. The last one was ontarget and focussed on what we did when the engine got mature. This time we zoom in on developments that go a bit beyond what we originally planned. One can argue that for instance some of the math extensions should have ended up here but for us a turning point was when additional par passes became stable, which was around the time of the 2024 ConTEXt meeting. We'll see what comes after that.

Hans Hagen
Hasselt NL
August 2024++

5 Introduction

2 A new take on paragraphs

Appears first in TUGboat.

7 A new take on paragraphs

3 Twins

Appears first in TUGboat.

9 Twins

4 Namespaces

Occasionally on TEX related mailing lists, meetings, articles or forums performance comes up. It makes no sense for me to go into the specific (assumed) bottlenecks mentioned but as in ConTEXt we do keep an eye on performance every now and then I also spend words on it, so here are some.

The nature of the (multilingual) user interface of ConTEXt there is extensive use of the `\csname` and related primitives. For instance, if we have the namespace `999>` and a keyword `testkeyword`, we can have a specific property set with:

```
\expandafter\def\csname 999>testkeyword\endcsname{}
```

We can then test if a macro with the inaccessible name '999>testkeyword' exists and has been set with a test command available in all engines that carry ε-TEX extensions:

```
\ifcsname 999>testkeyword\endcsname
    % whatever
\fi
```

In order to test this, the list of tokens starting at `9` and ending at `d` has to be converted into a (C) string that is used for a hash lookup. One can expect this to be a costly operation. In a 300 page book with many thousands of formulas this easily runs into the millions. Testing this five times on one million such tests gives:

```
0.303 0.293 0.283 0.301 0.298
```

for LuaMetaTEX and

```
0.276 0.287 0.287 0.274 0.274
```

for LuaTEX. I deliberately show five numbers because one has to keep some system load into account. When I'm interested in performance I only care about trends because no run ever gets the whole machine for its job. That said, where does the noticeable difference between these engines come from? It can partly be explained by LuaMetaTEX having more primitives and therefore a bit more overhead (more scattered code in memory and cpu cache). But as the basic code that kicks in here is not that much different I figured that it might be the hash lookup and, because indeed we had a follow up lookup in the hash (two steps), by using a larger hash table we could limit that to a direct hit.

```
0.288 0.281 0.280 0.288 0.277
```

So we ended up with similar measurements for these engines. Before we carry on, let's ask ourselves if these numbers worry us. Say that this book takes 12 seconds to process, does it matter much if we half this overhead? Probably not, but in the following, we need to keep in mind that much can interfere. A simple million times test is likely very cpu cache friendly. There are however other factors in play: convenience coding, abstraction, less cluttered tracing, more detailed feedback from the engine, less code and memory usage, the size of the format file. Trying to get lower numbers is also kind of fun.

Back to the user interface, we now introduce some abstraction (the `test` in the names avoids clashes with existing definitions):

```
\def\??testfoo    {999>}
\def\c!testkeyword{keyword}

\ifcsname\??testfoo\c!testkeyword\endcsname
    % whatever
\fi
```

Again LuaMetaTEX is a little slower but it is kind of noise:

```
0.243 0.243 0.247 0.241 0.249 luatex
0.251 0.250 0.250 0.249 0.249 luametatex
```

But how about the following timings for LuaMetaTEX:

```
0.136 0.143 0.139 0.139 0.140
0.132 0.132 0.133 0.129 0.130
```

In the first case we defined the namespace and keyword as follows:

```
\cdef\??testfoo    {999>}
\cdef\c!testkeyword{keyword}
```

A `\cdef`'s macro is basically an `\edef`. This definition is scanned as token list and therefore we know the macro has no arguments. It operates as any macro but in a `\csname` related command it is just passes as-is and only expanded when we need to do a lookup. When that happens we don't need to go through a token list (copy) but directly can go to string characters.

The second measurement shows a little improvement and is the outcome from an experiment with build in namespaces. Think of this:

```
\namespaceifcsnamedef\iftestfoocsname 999
```

```
\iftestfoocsname\c!testkeyword\endcsname
    % whatever
\fi
```

That variant is faster but we're talking .05 second on 2.5 million calls in the book because we already use `\cdef`. Even more important is to notice that most documents have only tens of thousands such calls. And 0.15 seconds csname "test and call" on the whole run is not that bad. So, if we go beyond `\cdef` usage we don't need the efficiency argument but the other ones. So, after a few days of playing with this I rejected this solution. First of all the source didn't become more readable. We also had many more commands because there were for instance:

```
\namespacecsnamedef      \csnamefoo      999
\namespacedefcsnamedef   \defcsnamefoo   999
\namespaceifcsnamedef    \ifcsnamefoo    999
\namespacebegincsnamedef\begincsnamefoo 999
```

We also had a callback for reporting associated names when tracing. Of course there can be use cases where we have tens of millions of `\csname` calls but I still need to find them. But don't expect miracles now that we're in these low numbers. Integrating all this is also not that trivial because TEX has two separated code paths for expandable commands and ones more related to housekeeping and typesetting (the mail loop). This means that one has to intercept expansion of encoded namespaces and that gives a bit of a mess, especially because we also need to handle nested csnames.

As an aside I also played a bit with 'compiling' regular csname commands followed by a namespace into one token but that was even more messier.[1] So in the end I removed all that experimental namespace code and happily accept the fact that there's nothing to gain, but it was a fun experiment.

As a side effect of this experiment I decided to enable a primitive that had been commented. When it was tested years ago there was no real gain but I realized that it could be implemented a bit more efficient in specific scenarios. Think of this:

```
\csname\ifcsname999>foobar:width\endcsname999>foo:width\fi\endcsname
```

when abstracted becomes:

```
\csname\ifcsname\??testme foobar:\c!width\endcsname\??testme
                                        foo:\c!width\fi\endcsname
```

[1] Occasionally I consider some compilation of tokens lists into more efficient ones but so far I could resist.

In both cases the same list of tokens (`\??testme foobar:\c!width`) has to be converted into a byte string, which we can avoid by:

```
\csname\ifcsname\??testme
                foobar:\c!width\endcsname\csnamestring\fi\endcsname
```

when we have a hit. After all, the found macro has a known name that has been registered as a string. This variant runs over 10 percent faster, which of course can be neglected, especially if we don't call it millions of times; the book has 400.000 calls to `\csnamestring`. But as with many optimizations: gaining 20 times 0.1 seconds on different subsystems eventually adds up to 20 % on a 10 seconds run for that 300 page, math extensive, book.

When looking at timings one always need to keep in mind that a simple test (in a loop) is very easy on the cpu cache while in a real document there can be more cache misses simply because the cache is limited in size. That is why in practice we often see a bit more positive impact than shown here. In the case of the `\csnamestring` we not only gain a bit on parameter handling but also on some font related operations, but again the gain depends on how many (more complex) font switches happen, which is more likely in for instance manuals.