LuaTEX 0.90 backend changes for PDF and more

Hans Hagen

Abstract

LuaTEX 0.90 brings a sweeping reorganization of the backend, especially PDF-related features. Many fundamental primitives, both commands and variables, have been renamed and generalized. Some primitives have been removed from the TEX interface, but can be made available through Lua. Internally, some uses of whatsits have been converted to regular nodes.

1 Introduction

The original design of TEX has a clear separation between the frontend and backend code. In principle, shipping out a page boils down to traversing the to-be-shipped-out box and translating the glyph, rule, glue, kern and list nodes into positioning just glyphs and rules on a canvas. The DVI backend is therefore relatively simple, as the DVI output format delegates to other programs the details of font inclusion and such into the final format; it just describes the pages.

Because we eventually want color and images as well, there is a mechanism to pass additional information to post-processing programs. One can insert `\special`s with directives like `insert image named foo.jpg`. The frontend as well as the backend are not concerned with what goes into a special; the DVI post-processor of course is.

The PDF backend, on the other hand, is more complex as it immediately produces the final typeset result and, as such, offers possibilities to insert verbatim code (`\pdfliteral`), images (`\pdfximage` cum suis), annotations, destinations, threads and all kinds of objects, reuse typeset content (`\pdfxform` cum suis); in the end, there are all kinds of `\pdf...` commands. The way these were implemented in LuaTEX prior to 0.82 violates the separation between frontend and backend, an inheritance from pdfTEX. Additional features such as protrusion and expansion add to that entanglement. However, because PDF is an evolving standard, occasionally we need to adapt the related code. A separation of code makes sure that the frontend can become stable (and hopefully frozen) at some point.[1]

In LuaTEX we had already started making this separation of specialized code, such as a cleaner implementation of font expansion, but all these `\pdf...` commands were still pervasive, leading to fuzzy dependencies, checks for backend modes, etc. so a logical step was to straighten all this out. That way we give LuaTEX a cleaner core constructed from traditional TEX, extended with ε-TEX, Aleph/Omega, and LuaTEX functionality.

2 Extensions

A first step, then, was to transform generic (i.e. independent from the backend) functionality which was still (sort of) bound to Aleph and pdfTEX, into core functionality. A second step was to reorganize the backend specific PDF code, i.e. move it out of the core and into the group of extension commands. This extension group is somewhat special and originates in traditional TEX; it is the way to add your own functionality to TEX, the program.

As an example for future programmers, Don Knuth added four (connected) primitives as extensions: `\openout`, `\closeout`, `\write` and `\special`. The Aleph and pdfTEX engines, on the other hand, put some functionality in extensions and some in the core. This arose from the fact that dealing with variables in extensions is often inconvenient, as they are then seen as (unexpandable) commands instead of integers, token lists, etc. That the write-related commands are there is almost entirely due to being the demonstration of the mechanism; everything related to *reading* files is in the core. There is one property that perhaps forces us to keep the writers there, and that's the `\immediate` prefix.[2]

In the process of separating, we reshuffled the code base a bit; the current use of the extensions mechanism still suits as an example and also gives us backward compatibility. However, new backend primitives will not be added there but rather in specific plugins (if needed at all).

3 From whatsits to nodes: images, forms, directions

The PDF backend introduced two new concepts into the core: (reusable) images and (reusable) content (wrapped in boxes). In keeping with good TEX practice, these were implemented as whatsits (a node type for extensions); but this created, as a side effect, an anomaly in the handling of such nodes. Consider looping over a node list where we need to check dimensions of nodes; in Lua, we can write something like this:

```
while n do
    if n.id == glyph then
```

Author's note: Thanks to Alan Braslau and Karl Berry for text corrections.

[1] In practice nowadays, the backend code changes little, because the PDF produced by LuaTEX is rather simple and is easily adapted to the changing standard.

[2] Unfortunately we're stuck with `\immediate` in the backend; a `deferred` keyword would have been handier, especially since other backend-related commands can also be immediate.

```
        -- wd ht dp
    elseif n.id == rule then
        -- wd ht dp
    elseif n.id == kern then
        -- wd
    elseif n.id == glue then
        -- size stretch shrink
    elseif n.id == whatsits then
        if n.subtype == pdfxform then
            -- wd ht dp
        elseif n.subtype == pdfximage then
            -- wd ht dp
        end
    end
    n = n.next
end
```

So for each node in the list, we need to check these two whatsit subtypes. But as these two concepts are rather generic, there is no evident need to implement it this way. Of course the backend has to provide the inclusion and reuse, but the frontend can be agnostic about this. That is, at the input end, in specifying these two injects, we only have to make sure we pass the right information (so the scanner might differentiate between backends).

Thus, in LuaTEX these two concepts have been promoted to core features:

```
\pdfxform           \saveboxresource
\pdfximage          \saveimageresource
\pdfrefxform        \useboxresource
\pdfrefximage       \useimageresource
\pdflastxform       \lastsavedboxresourceindex
\pdflastximage      \lastsavedimageresourceindex
\pdflastximagepages \lastsavedimageresourcepages
```

The index should be considered an arbitrary number set to whatever the backend plugin decides to use as an identifier. These are no longer whatsits, but a special type of rule; after all, TEX is only interested in dimensions. Given this change, the previous code can be simplified to:

```
while n do
    if n.id == glyph then
        -- wd ht dp
    elseif n.id == rule then
        -- wd ht dp
    elseif n.id == kern then
        -- wd
    elseif n.id == glue then
        -- size stretch shrink
    end
    n = n.next
end
```

The only consequence for the previously existing rule type (which, in fact, is also something that needs to be dealt with in the backend, depending on the target format) is that a normal rule now has subtype 0 while the box resource has subtype 1 and the image subtype 2.

If a package writer wants to retain the pdfTEX names, the previous table can be used; just prefix `\let`. For example, the first line would be (spaces optional, of course):

```
\let \pdfxform \saveboxresource
```

3.1 Direction nodes

A similar change has been made for "direction" nodes, which were also previously whatsits. These are now normal nodes so again, instead of consulting whatsit subtypes, we can now just check the id of a node.

It should be apparent that all of these changes from whatsits to normal nodes already greatly simplify the code base.

4 Commands promoted to the core

Many more commands have been promoted to the core. Here is an additional list of original pdfTEX commands and their new counterparts (this time with the `\let` included):

```
\let\pdfpagewidth       \pagewidth
\let\pdfpageheight      \pageheight

\let\pdfadjustspacing   \adjustspacing
\let\pdfprotrudechars   \protrudechars
\let\pdfnoligatures     \ignoreligaturesinfont
\let\pdffontexpand      \expandglyphsinfont
\let\pdfcopyfont        \copyfont

\let\pdfnormaldeviate   \normaldeviate
\let\pdfuniformdeviate  \uniformdeviate
\let\pdfsetrandomseed   \setrandomseed
\let\pdfrandomseed      \randomseed

\let\ifpdfabsnum        \ifabsnum
\let\ifpdfabsdim        \ifabsdim
\let\ifpdfprimitive     \ifprimitive

\let\pdfprimitive       \primitive

\let\pdfsavepos         \savepos
\let\pdflastxpos        \lastxpos
\let\pdflastypos        \lastypos

\let\pdftexversion      \luatexversion
\let\pdftexrevision     \luatexrevision
\let\pdftexbanner       \luatexbanner

\let\pdfoutput          \outputmode
\let\pdfdraftmode       \draftmode

\let\pdfpxdimen         \pxdimen

\let\pdfinsertht        \insertht
```

```
\protected\def\pdfliteral        {\pdfextension literal }
\protected\def\pdfcolorstack     {\pdfextension colorstack }
\protected\def\pdfsetmatrix      {\pdfextension setmatrix }
\protected\def\pdfsave           {\pdfextension save\relax}
\protected\def\pdfrestore        {\pdfextension restore\relax}
\protected\def\pdfobj            {\pdfextension obj }
\protected\def\pdfrefobj         {\pdfextension refobj }
\protected\def\pdfannot          {\pdfextension annot }

\protected\def\pdfstartlink      {\pdfextension startlink }
\protected\def\pdfendlink        {\pdfextension endlink\relax}
\protected\def\pdfoutline        {\pdfextension outline }
\protected\def\pdfdest           {\pdfextension dest }
\protected\def\pdfthread         {\pdfextension thread }
\protected\def\pdfstartthread    {\pdfextension startthread }
\protected\def\pdfendthread      {\pdfextension endthread\relax}

\protected\def\pdfinfo           {\pdfextension info }
\protected\def\pdfcatalog        {\pdfextension catalog }
\protected\def\pdfnames          {\pdfextension names }
\protected\def\pdfincludechars   {\pdfextension includechars }
\protected\def\pdffontattr       {\pdfextension fontattr }
\protected\def\pdfmapfile        {\pdfextension mapfile }
\protected\def\pdfmapline        {\pdfextension mapline }
\protected\def\pdftrailer        {\pdfextension trailer }
\protected\def\pdfglyphtounicode {\pdfextension glyphtounicode }
```

Table 1: List of pdfTEX commands and their new `\pdfextension` equivalents in LuaTEX.

5 Commands: from `\pdf...` to `\pdfextension`

There are many commands that start with `\pdf` and, over the history of development of pdfTEX and LuaTEX, some have been added, some have been renamed, others removed. Instead of the many, we now have just one: `\pdfextension`. A couple of usage examples:

```
\pdfextension literal {1 0 0 2 0 0 cm}
\pdfextension obj     {/foo (bar)}
```

Here, we pass a keyword that tells for what to scan and what to do with it. A backward-compatible interface is easy to write. Although it delegates a bit more management of these `\pdf` commands to the macro package, the responsibility for dealing with such low-level, error-prone calls is there anyway. The full list of `\pdfextension`s is given in table 1. The scanning after the keyword is the same as for pdfTEX.

6 Variables: from `\pdf...` to `\pdfvariable`

As with commands, there are many variables that can influence the PDF backend. The most important one was, of course, that which set the output mode (`\pdfoutput`). Well, that one is gone and has been replaced by `\outputmode`. A value of 1 means that we produce PDF.

One complication of variables is that (if we want to be compatible), we need to have them as real TEX registers. However, as most of them are optional, an easy way out is simply not to define them in the engine. In order to be able to still deal with them as registers (which is backward compatible), we define them as shown in table 2.

You can set them as follows (the values shown here are the initial values):

```
\pdfcompresslevel          9
\pdfobjcompresslevel       1
\pdfdecimaldigits          3
\pdfgamma               1000
\pdfimageresolution       71
\pdfimageapplygamma        0
\pdfimagegamma          2200
\pdfimagehicolor           1
\pdfimageaddfilename       1
\pdfpkresolution          72
\pdfinclusioncopyfonts     0
\pdfinclusionerrorlevel    0
\pdfignoreunknownimages    0
\pdfreplacefont            0
\pdfgentounicode           0
\pdfpagebox                0
\pdfminorversion           4
\pdfuniqueresname          0
```

```
\edef\pdfminorversion        {\pdfvariable minorversion}
\edef\pdfcompresslevel       {\pdfvariable compresslevel}
\edef\pdfobjcompresslevel    {\pdfvariable objcompresslevel}
\edef\pdfdecimaldigits       {\pdfvariable decimaldigits}

\edef\pdfhorigin             {\pdfvariable horigin}
\edef\pdfvorigin             {\pdfvariable vorigin}

\edef\pdfgamma               {\pdfvariable gamma}
\edef\pdfimageresolution     {\pdfvariable imageresolution}
\edef\pdfimageapplygamma     {\pdfvariable imageapplygamma}
\edef\pdfimagegamma          {\pdfvariable imagegamma}
\edef\pdfimagehicolor        {\pdfvariable imagehicolor}
\edef\pdfimageaddfilename    {\pdfvariable imageaddfilename}
\edef\pdfignoreunknownimages {\pdfvariable ignoreunknownimages}

\edef\pdfinclusioncopyfonts  {\pdfvariable inclusioncopyfonts}
\edef\pdfinclusionerrorlevel {\pdfvariable inclusionerrorlevel}
\edef\pdfpkmode              {\pdfvariable pkmode}
\edef\pdfpkresolution        {\pdfvariable pkresolution}
\edef\pdfgentounicode        {\pdfvariable gentounicode}

\edef\pdflinkmargin          {\pdfvariable linkmargin}
\edef\pdfdestmargin          {\pdfvariable destmargin}
\edef\pdfthreadmargin        {\pdfvariable threadmargin}
\edef\pdfformmargin          {\pdfvariable formmargin}

\edef\pdfuniqueresname       {\pdfvariable uniqueresname}
\edef\pdfpagebox             {\pdfvariable pagebox}
\edef\pdfpagesattr           {\pdfvariable pagesattr}
\edef\pdfpageattr            {\pdfvariable pageattr}
\edef\pdfpageresources       {\pdfvariable pageresources}
\edef\pdfxformattr           {\pdfvariable xformattr}
\edef\pdfxformresources      {\pdfvariable xformresources}
```

Table 2: List of pdfTEX variables and their new `\pdfvariable` equivalents in LuaTEX.

```
\pdfhorigin             1in
\pdfvorigin             1in
\pdflinkmargin          0pt
\pdfdestmargin          0pt
\pdfthreadmargin        0pt
```

Their removal from the frontend has helped again to clean up the code and, by making them registers, their use is still compatible. A call to `\pdfvariable` defines an internal register that keeps the value (of course this value can also be influenced by the backend itself). Although they are real registers, they live in a protected namespace:

```
\meaning\pdfcompresslevel
```

which gives:

```
macro:->[internal backend integer]
```

It's perhaps unfortunate that we have to remain compatible because a setter and getter would be much nicer. I am still considering writing the extension primitive in Lua using the token scanner, but it might not be possible to remain compatible then. This is not so much an issue for ConTEXt that always has had backend drivers, but, rather, for other macro packages that have users expecting the primitives (or counterparts) to be available.

7 Read-only variables: from `\pdf...` to `\pdffeedback`

The backend can report on some properties that were also accessible via `\pdf...` primitives. Because these are read-only variables, another primitive now handles them: `\pdffeedback`. This primitive can be used to define compatible alternatives, as shown in table 3.

The variables are internal, so they are anonymous. When we ask for the meaning of some that were previously defined:

```
\meaning\pdfhorigin
\meaning\pdfcompresslevel
\meaning\pdfpageattr
```

```
\def\pdfcolorstackinit {\pdffeedback colorstackinit}
\def\pdfcreationdate   {\pdffeedback creationdate}
\def\pdffontname       {\numexpr\pdffeedback fontname\relax}
\def\pdffontobjnum     {\numexpr\pdffeedback fontobjnum\relax}
\def\pdffontsize       {\dimexpr\pdffeedback fontsize\relax}
\def\pdflastannot      {\numexpr\pdffeedback lastannot\relax}
\def\pdflastlink       {\numexpr\pdffeedback lastlink\relax}
\def\pdflastobj        {\numexpr\pdffeedback lastobj\relax}
\def\pdfpageref        {\numexpr\pdffeedback pageref\relax}
\def\pdfretval         {\numexpr\pdffeedback retval\relax}
\def\pdfxformname      {\numexpr\pdffeedback xformname\relax}
```

Table 3: List of read-only pdfTEX variables and their new `\pdffeedback` equivalents in LuaTEX.

we will get, similar to the above:

```
macro:->[internal backend dimension]
macro:->[internal backend integer]
macro:->[internal backend tokenlist]
```

8 `\pdf...` primitives removed

Finally, here is the list of primitives that have been removed, with no TEX-level equivalent available. Many were experimental, and they can be easily be provided to TEX using Lua.

```
\knaccode
\knbccode
\knbscode
\pdfadjustinterwordglue
\pdfappendkern
\pdfeachlinedepth
\pdfeachlineheight
\pdfelapsedtime
\pdfescapehex
\pdfescapename
\pdfescapestring
\pdffiledump
\pdffilemoddate
\pdffilesize
\pdffirstlineheight
\pdfforcepagebox
\pdfignoreddimen
\pdflastlinedepth
\pdflastmatch
\pdflastximagecolordepth
\pdfmatch
\pdfmdfivesum
\pdfmovechars
\pdfoptionalwaysusepdfpagebox
\pdfoptionpdfinclusionerrorlevel
\pdfprependkern
\pdfresettimer
\pdfshellescape
\pdfsnaprefpoint
\pdfsnapy
\pdfsnapycomp
\pdfstrcmp
\pdfunescapehex
\pdfximagebbox
\shbscode
\stbscode
```

9 Conclusion

The advantage of a clean backend separation, supported by just the three primitives `\pdfextension`, `\pdfvariable` and `\pdffeedback`, as well as a collection of registers, is that we can now further clean the code base, which remains a curious mix of combined engine code, sometimes and sometimes not converted to C from Pascal. A clean separation also means that if someone wants to tune the backend for a special purpose, the frontend can be left untouched. We will get there eventually.

All the definitions shown here are available in the file `luatex-pdf.tex`, which is part of the ConTEXt distribution.

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com