LuaTEX: Deeply nested notes

Hans Hagen

1 Introduction

One of the mechanisms that is not on a user's retina when he or she starts using TEX is 'inserts'. An insert is material that is entered at one point but will appear somewhere else in the output. Footnotes for instance can be implemented using inserts. You create a reference symbol in the running text and put note text at the bottom of the page or at the end of a chapter or document. But as you don't want to do that moving around of notes yourself TEX provides macro writers with the inserts mechanism that will do some of the housekeeping. Inserts are quite clever in the sense that they are taken into account when TEX splits off a page. A single insert can even be split over two or more pages.

Other examples of inserts are floats that move to the top or bottom of the page depending on requirements and/or available space. Of course the macro package is responsible for packaging such a float (for instance an image) but by finally putting it in an insert TEX itself will attempt to deal with accumulated floats and help you move kept over floats to following pages. When the page is finally assembled (in the output routine) the inserts for that page become available and can be put at the spot where they belong. In the process TEX has made sure that we have the right amount of space available.

However, let's get back to notes. In ConTEXt we can have many variants of them, each taken care of by its own class of inserts. This works quite well — as long as a note is visible for TEX, which more or less means: ends up in the main page flow. Consider the following situation:

```
before \footnote{the note} after
```

When the text is typeset, a symbol is placed directly after `before` and the note itself ends up at the bottom of the page. It also works when we wrap the text in an horizontal box:

```
\hbox{before \footnote{the note} after}
```

But it fails as soon as we go further:

```
\hbox{\hbox{before \footnote{the note} after}}
```

Here we get the reference but no note. This also fails:

```
\vbox{before \footnote{the note} after}
```

Can you imagine what happens if we do the following? (In this ConTEXt table, `\NC` separates columns and `\NR` separates rows.)

```
\starttabulate
\NC knuth \NC test \footnote{knuth}
    \input knuth \NC \NR
\NC tufte \NC test \footnote{tufte}
    \input tufte \NC \NR
\NC ward  \NC test \footnote{ward}
    \input ward \NC \NR
\stoptabulate
```

This mechanism uses alignments as well as quite some boxes. The paragraphs are nicely split over pages but still appear as boxes to TEX which make inserts invisible. Only the three reference symbols would remain visible. But because in ConTEXt we know when notes tend to disappear, we take some provisions, and contrary to what you might expect the notes actually do show up. However, they are flushed in such a way that they end up on the page where the table ends. Normally this is no big deal as we will often use local notes that end up at the end of the table instead of the bottom of the page, but still.

The mechanism to deal with notes in ConTEXt is somewhat complex at the source code level. To mention a few properties we have to deal with:

- Notes are collected and can be accessed any time.
- Notes are flushed either directly or delayed.
- Notes can be placed anywhere, any time, perhaps in subsets.
- Notes can be associated with lines in paragraphs.
- Notes can be placed several times with different layouts.

So, we have some control over flushing and placement, but real synchronization between for instance table entries having notes and the note content ending up on the same page is impossible.

Within the LuaTEX team we have been discussing more control over inserts and we will definitely deal with that in upcoming releases as more control is needed for complex multi-column document layouts. But as we have some other priorities these extensions have to wait.

As a prelude to them I experimented a bit with making these deeply buried inserts visible. Of course I use Lua for this as TEX itself does not provide the kind of access we need for this kind of manipulation.

2 Deep down inside

Say that we have the following boxed footnote. How does it end up in LuaTEX?

```
\vbox{a\footnote{b}c}
```

Actually it depends on the macro package but the principles remain the same. In LuaTEX 0.50 and the ConTEXt version used at the time of this writing we get a (nested) linked list that prints as follows:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
    <node   30 <  611 >  580 : whatsit 6>
    <node  611 <  580 >  493 : hlist 0>
    <node  580 <  493 >  653 : glyph 256>
    <node  493 <  653 >  797 : penalty 0>
    <node  653 <  797 >  424 : kern 1>
    <node  797 <  424 >  826 : hlist 2>
      <node  445 <  563 >  nil : hlist 2>
        <node  420 <  817 >  821 : whatsit 35>
        <node  817 <  821 >  nil : glyph 256>
    <node  507 <  826 > 1272 : kern 1>
    <node  826 < 1272 > 1333 : glyph 256>
    <node 1272 < 1333 >  830 : penalty 0>
    <node 1333 <  830 >  888 : glue 15>
    <node  830 <  888 >  nil : glue 9>
  <node  838 <  507 >  nil : ins 131>
```

The numbers are internal references to the node memory pool. Each line represents a node:

```
<node prev_index < index > next_index : type subtype>
```

The whatsits carry directional information and the deeply nested hlist is the note symbol. If we forget about whatsits, kerns and penalties, we can simplify the listing to:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
    <node  580 <  493 >  653 : glyph 256>
    <node  797 <  424 >  826 : hlist 2>
      <node  445 <  563 >  nil : hlist 2>
        <node  817 <  821 >  nil : glyph 256>
    <node  826 < 1272 > 1333 : glyph 256>
  <node  838 <  507 >  nil : ins 131>
```

So, we have a vlist (the \vbox), which has one line being a hlist. Inside we have a glyph (the ‘a’) followed by the raised symbol (the ‘^{1}’) and next comes the second glyph (the ‘b’). But watch how the insert ends up at the end of the line. Although the insert will not show up in the document, it sits there waiting to be used. So we have:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
  <node  838 <  507 >  nil : ins 131>
```

but we need:

```
<node   26 <  862 >  nil : vlist 0>
  <node  401 <  838 >  507 : hlist 1>
<node  838 <  507 >  nil : ins 131>
```

Now, we could use the fact that inserts end up at the end of the line, but as we need to recursively identify them anyway, we cannot actually use this fact to optimize the code.

In case you wonder how multiple inserts look, here is an example:

```
\vbox{a\footnote{b}\footnote{c}d}
```

This boils down to:

```
<node   26 < 1324 >  nil : vlist 0>
  <node  401 < 1348 >  507 : hlist 1>
  <node 1348 <  507 >  457 : ins 131>
  <node  507 <  457 >  nil : ins 131>
```

And in case you wonder what more can end up at the end, vertically adjusted material (\vadjust) as well as marks (\mark) also get this treatment.

```
\vbox{a\footnote{b}\vadjust{c}%
      \footnote{d}e\mark{f}}
```

As you see, we start with the line itself, followed by a mixture of inserts and vertical adjusted content (that will be placed before that line). This trace also shows the list 2 levels deep.

```
<node   26 < 1324 >  nil : vlist 0>
  <node  401 < 1348 >  507 : hlist 1>
  <node 1348 <  507 >  862 : ins 131>
  <node  507 <  862 >  240 : hlist 1>
  <node  862 <  240 > 2288 : ins 131>
  <node  240 < 2288 >  nil : mark 0>
```

Currently vadjust nodes have the same subtype as an ordinary hlist but in LuaTEX versions beyond 0.50 they will have a dedicated subtype.

We can summarize the pattern of one ‘line’ in a vertical list as:

```
[hlist][insertmarkvadjust]*[penaltyglue]+
```

In case you wonder what happens with for instance specials, literals (and other whatsits): these end up in the hlist that holds the line. Only inserts, marks and vadjusts migrate to the outer level, but as they stay inside the vlist, they are not visible to the page builder unless we’re dealing with the main vertical list. Compare:

```
this is a regular paragraph possibly with
inserts and they will be visible as the lines
are appended to the main vertical list \par
```

with:

```
but \vbox {this is a nested paragraph where
inserts will stay with the box} and not migrate
here \par
```

So much for the details; let’s move on to how we can get around this phenomenon.

3 Some LuaTEX magic

The following code is just the first variant I made; ConTEXt ships with a more extensive variant. Also, in ConTEXt this is part of a larger suite of manipulative actions but it does not make much sense (at least not now) to discuss this framework here.

We start with defining a couple of convenient shortcuts.

```
local hlist = node.id('hlist')
local vlist = node.id('vlist')
local ins   = node.id('ins')
```

We can write a more compact solution but splitting up the functionality better shows what we're doing. The main migration function hooks into the callback `build_page`. Unlike other callbacks that do phases in building lists and pages this callback does not expect the head of a list as argument. Instead, we operate directly on the additions to the main vertical list which is accessible as `tex.lists.contrib_head`.

```
local deal_with_inserts -- forward reference

local function migrate_inserts(where)
    local current = tex.lists.contrib_head
    while current do
        local id = current.id
        if id == vlist or id == hlist then
           current = deal_with_inserts(current)
        end
        current = current.next
    end
end

callback.register('buildpage_filter',
                  migrate_inserts)
```

So, effectively we scan for vertical and horizontal lists and deal with embedded inserts when we find them. In ConTEXt the migratory function is just one of the functions that is applied to this filter.

We locate inserts and collect them in a list with `first` and `last` as head and tail and do so recursively. When we have run into inserts we insert them after the horizontal or vertical list that had embedded them.

```
local locate -- forward reference

deal_with_inserts = function(head)
    local h, first, last = head.list, nil, nil
    while h do
        local id = h.id
        if id == vlist or id == hlist then
          h, first, last = locate(h,first,last)
        end
        h = h.next
    end
    if first then
        local n = head.next
        head.next = first
        first.prev = head
        if n then
            last.next = n
            n.prev = last
        end
        return last
    else
        return head
    end
end
```

The `locate` function removes inserts and adds them to a new list, that is passed on down in recursive calls and eventually is returned back to the caller.

```
locate = function(head,first,last)
  local current = head
  while current do
    local id = current.id
    if id == vlist or id == hlist then
      current.list, first, last
        = locate(current.list,first,last)
      current = current.next
    elseif id == ins then
      local insert = current
      head, current = node.remove(head,current)
      insert.next = nil
      if first then
        insert.prev = last
        last.next = insert
      else
        insert.prev = nil
        first = insert
      end
      last = insert
    else
      current = current.next
    end
  end
  return head, first, last
end
```

As we can encounter the content several times in a row, it makes sense to mark already processed inserts. This can for instance be done by setting an attribute. Of course one has to make sure that this attribute is not used elsewhere.

```
if not node.has_attribute(current,8061) then
    node.set_attribute(current,8061,1)
    current = deal_with_inserts(current)
end
```

Or integrated:

```
local has_attribute = node.has_attribute
local set_attribute = node.set_attribute
local function migrate_inserts(where)
  local current = tex.lists.contrib_head
  while current do
    local id = current.id
    if id == vlist or id == hlist then
      if has_attribute(current,8061) then
        -- maybe some tracing message
      else
        set_attribute(current,8061,1)
        current = deal_with_inserts(current)
      end
    end
    current = current.next
  end
end; callback.register('buildpage_filter',
                       migrate_inserts)
```

4 A few remarks

Surprisingly, the amount of code needed for insert migration is not that large. This makes one wonder why TEX does not provide this feature itself as it could have saved macro writers quite some time and headaches. Performance can be a reason, unpredictable usage and side effects might be another. Only one person knows the answer.

In ConTEXt this mechanism is built in and it can be enabled by saying:

```
\automoveinserts
```

Future versions of ConTEXt will do this automatically and also provide some control over what classes of inserts are moved around. We will probably overhaul the note handling mechanism a few more times anyway as LuaTEX evolves due especially to the demands from critical editions, which use many kind of notes.

5 Summary of code

The following code should work in plain LuaTEX:

```
\directlua 0 {
local hlist         = node.id('hlist')
local vlist         = node.id('vlist')
local ins           = node.id('ins')
local has_attribute = node.has_attribute
local set_attribute = node.set_attribute

local status = 8061

local function locate(head,first,last)
  local current = head
  while current do
    local id = current.id
    if id == vlist or id == hlist then
      current.list, first, last
        = locate(current.list,first,last)
      current = current.next
    elseif id == ins then
      local insert = current
      head, current = node.remove(head,current)
      insert.next = nil
      if first then
        insert.prev, last.next = last, insert
      else
        insert.prev, first = nil, insert
      end
      last = insert
    else
      current = current.next
    end
  end
  return head, first, last
end

local function migrate_inserts(where)
  local current = tex.lists.contrib_head
  while current do
    local id = current.id
    if id == vlist or id == hlist and
       not has_attribute(current,status) then
      set_attribute(current,status,1)
      local h, first, last = current.list, nil, nil
      while h do
        local id = h.id
        if id == vlist or id == hlist then
          h, first, last = locate(h,first,last)
        end
        h = h.next
      end
      if first then
        local n = current.next
        if n then
          last.next, n.prev = n, last
        end
        current.next, first.prev = first, current
        current = last
      end
    end
    current = current.next
  end
end

callback.register('buildpage_filter',
                  migrate_inserts)
}
```

Alternatively you can put the code in a file and load that with:

```
\directlua {require "luatex-inserts.lua"}
```

A simple plain test is:

```
\vbox{a\footnote{1}{1}b}
\hbox{a\footnote{2}{2}b}
```

The first footnote only shows up when we have hooked our migrator into the callback. Not a bad result for 60 lines of Lua code.

◇ Hans Hagen
Pragma ADE
The Netherlands
http://luatex.org