low level TEX

loops

Contents

1 Introduction

I have hesitated long before I finally decided to implement native loops in LuaMetaTEX. Among the reasons against such a feature is that one can define macros that do loops (preferably using tail recursion). When you don't need an expandable loop, counters can be used, otherwise there are dirty and obscure tricks that can be of help. This is often the area where tex programmers can show off but the fact remains that we're using side effects of the expansion machinery and specific primitives like `\romannumeral` magic. In LuaMetaTEX it is actually possible to use the local control mechanism to hide loop counter advance and checking but that comes with at a performance hit. And, no matter what tricks one used, tracing becomes pretty much cluttered.

In the next sections we describe the new native loop primitives in LuaMetaTEX as well as the more traditional ConTEXt loop helpers.

2 Primitives

Because MetaPost, which is also a macro language, has native loops, it makes sense to also have native loops in TEX and in LuaMetaTEX it was not that hard to add it. One variant uses the local control mechanism which is reflected in its name and two others collect expanded bodies. In the local loop content gets injected as we go, so this one doesn't work well in for instance an `\edef`. The macro takes the usual three loop numbers as well as a token list:

```
\localcontrolledloop 1 100000 1 {%
    % body
}
```

Here is an example of usage:

```
\localcontrolledloop 1 5 1 {%
    [\number\currentloopiterator]
    \localcontrolledloop 1 10 1 {%
        (\number\currentloopiterator)
```

```
    }%
    [\number\currentloopiterator]
    \par
}
```

The `\currentloopiterator` is a numeric token so you need to explicitly serialize it with `\number` or `\the` if you want it to be typeset:

[1] (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) [1]
[2] (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) [2]
[3] (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) [3]
[4] (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) [4]
[5] (1) (2) (3) (4) (5) (6) (7) (8) (9) (10) [5]

Here is another example. This time we also show the current nesting:

```
\localcontrolledloop 1 100 1 {%
    \ifnum\currentloopiterator>6\relax
        \quitloop
    \else
        [\number\currentloopnesting:\number\currentloopiterator]
        \localcontrolledloop 1 8 1 {%
            (\number\currentloopnesting:\number\currentloopiterator)
        }\par
    \fi
}
```

Watch the `\quitloop`: it will end the loop at the *next* iteration so any content after it will show up. Normally this one will be issued in a condition and we want to end that properly.

[1:1] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:2] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:3] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:4] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:5] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:6] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)

The three loop variants all perform differently:

```
l:\testfeatureonce {1000} {\localcontrolledloop 1 2000 1 {\relax}} %
  \elapsedtime
e:\testfeatureonce {1000} {\expandedloop        1 2000 1 {\relax}} %
```

```
  \elapsedtime
u:\testfeatureonce {1000} {\unexpandedloop      1 2000 1 {\relax}} %
  \elapsedtime
```

An unexpanded loop is (of course) the fastest because it only collects and then feeds back the lot. In an expanded loop each cycle does an expansion of the body and collects the result which is then injected afterwards, and the controlled loop just expands the body each iteration.

```
l: 0.093
e: 0.075
u: 0.024
```

The different behavior is best illustrated with the following example:

```
\edef\TestA{\localcontrolledloop 1 5 1 {A}} % out of order
\edef\TestB{\expandedloop        1 5 1 {B}}
\edef\TestC{\unexpandedloop      1 5 1 {C\relax}}
```

We can show the effective definition:

```
\meaningasis\TestA
\meaningasis\TestB
\meaningasis\TestC
```

```
A: \TestA
B: \TestB
C: \TestC
```

Watch how the first test pushes the content in the main input stream:

```
AAAAA
\def \TestA
\def \TestB BBBBB
\def \TestC C\relax C\relax C\relax C\relax C\relax
```

```
A:
B: BBBBB
C: CCCCC
```

Here are some examples that show what gets expanded and what not:

```
\edef\whatever
  {\expandedloop 1 10 1
```

```
     {(\number\currentloopiterator)
      \scratchcounter=\number\currentloopiterator\relax}}

\meaningasis\whatever
```

```
\def \whatever (1) \scratchcounter =1\relax (2) \scratchcounter =2\relax (3) \scratchcounter
=3\relax (4) \scratchcounter =4\relax (5) \scratchcounter =5\relax (6) \scratchcounter
=6\relax (7) \scratchcounter =7\relax (8) \scratchcounter =8\relax (9) \scratchcounter
=9\relax (10) \scratchcounter =10\relax
```

A local control encapsulation hides the assignment:

```
\edef\whatever
  {\expandedloop 1 10 1
     {(\number\currentloopiterator)
      \beginlocalcontrol
      \scratchcounter=\number\currentloopiterator\relax
      \endlocalcontrol}}

\meaningasis\whatever
```

```
\def \whatever (1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
```

Here we see the assignment being retained but with changing values:

```
\edef\whatever
  {\unexpandedloop 1 10 1
     {\scratchcounter=1\relax}}

\meaningasis\whatever
```

```
\def \whatever \scratchcounter =1\relax \scratchcounter =1\relax \scratchcounter =1\relax
\scratchcounter =1\relax \scratchcounter =1\relax \scratchcounter =1\relax \scratchcounter
=1\relax \scratchcounter =1\relax \scratchcounter =1\relax \scratchcounter =1\relax
```

We get no expansion at all:

```
\edef\whatever
  {\unexpandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}

\meaningasis\whatever
```

```
\def \whatever \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax
\scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter
=0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax
```

And here we have a mix:

```
\edef\whatever
  {\expandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}

\meaningasis\whatever
```

```
\def \whatever \scratchcounter =1\relax \scratchcounter =2\relax \scratchcounter =3\relax
\scratchcounter =4\relax \scratchcounter =5\relax \scratchcounter =6\relax \scratchcounter
=7\relax \scratchcounter =8\relax \scratchcounter =9\relax \scratchcounter =10\relax
```

There is one feature worth noting. When you feed three numbers in a row, like here, there is a danger of them being seen as one:

```
\expandedloop
  \number\dimexpr1pt
  \number\dimexpr2pt
  \number\dimexpr1pt
  {}
```

This gives an error because a too large number is seen. Therefore, these loops permit leading equal signs, as in assignments (we could support keywords but it doesn't make much sense):

```
\expandedloop =\number\dimexpr1pt =\number\dimexpr2pt =\number\dimexpr1pt{}
```

3 Wrappers

We always had loop helpers in ConTEXt and the question is: “What we will gain when we replace the definitions with ones using the above?”. The answer is: “We have little performance but not as much as one expects!”. This has to do with the fact that we support #1 as iterator and #2 as (verbose) nesting values and that comes with some overhead. It is also the reason why these loop macros are protected (unexpandable). However, using the primitives might look somewhat more natural in low level TEX code.

Also, replacing their definitions can have side effects because the primitives are (and will be) still experimental so it's typically a patch that I will run on my machine for a while.

Here is an example of two loops. The inner state variables have one hash, the outer one extra:

```
\dorecurse{2}{
    \dostepwiserecurse{1}{10}{2}{
        (#1:#2) [##1:##2]
    }\par
}
```

We get this:

(1:1) [1:2] (1:1) [3:2] (1:1) [5:2] (1:1) [7:2] (1:1) [9:2]
(2:1) [1:2] (2:1) [3:2] (2:1) [5:2] (2:1) [7:2] (2:1) [9:2]

We can also use two state macro but here we would have to store the outer ones:

```
\dorecurse {2} {
    /\recursedepth:\recurselevel/
    \dostepwiserecurse {1} {10} {2} {
        <\recursedepth:\recurselevel>
    }\par
}
```

That gives us:

/1:1/ <2:1> <2:3> <2:5> <2:7> <2:9>
/1:2/ <2:1> <2:3> <2:5> <2:7> <2:9>

An endless loop works as follows:

```
\doloop {
    ...
    \ifsomeconditionismet
        ...
        \exitloop
    \else
        ...
    \fi
  % \exitloopnow
    ...
}
```

Because of the way we quit there will not be a new implementation in terms of the loop primitives. You need to make sure that you don't leave in the middle of an ongoing condition. The second exit is immediate.

We also have a (simple) expanded variant:

```
\edef\TestX{\doexpandedrecurse{10}{!}} \meaningasis\TestX
```

This helper can be implemented in terms of the loop primitives which makes them a bit faster, but these are not critical:

\def \TestX !!!!!!!!!!

A variant that supports #1 is the following:

```
\edef\TestX{\doexpandedrecursed{10}{#1}} \meaningasis\TestX
```

So:

\def \TestX 12345678910

3 Colofon

Author	Hans Hagen
ConTEXt	2023.04.27 17:04
LuaMetaTEX	2.1008
Support	www.pragma-ade.com
	contextgarden.net