Beyond Trip and Trap: Testing the urtext WEB sources

David Fuchs

Abstract

Finding some undefined behavior and other tricky bugs in Donald Knuth's original WEB sources of TEX, METAFONT, &c, and a request for plain TEX and METAFONT input files.

1 Preparing for the next TEX tuneup

In anticipation of TEX and METAFONT approaching versions π and e, respectively, I've been working on some tools to help find any remaining "undefined behavior" bugs in the core code, so they can be addressed for posterity.

In the remainder of this note, I'll mostly refer to TEX for brevity, but everything said here applies to the set of principal Stanford WEB programs: Tangle, Weave, TEX, METAFONT, TFtoPL, PLtoTF, DVItype, and GFtype.

2 Range checks

As usual with this sort of tool, the first kind of "undefined behavior" to watch for at runtime is any attempt to read from an uninitialized variable. (One caveat, however: TEX and METAFONT do this intentionally at one place, with the *ready_already* variable.) This is straightforward to handle using an auxiliary bit per variable that indicates whether it has been written to yet. Each field in a record (C `struct`) gets its own bit, as they can be assigned to individually.

Next, each array has a declared range of valid index values, so each access to an array should check that the index is properly in range. Also, each read of an array element should check that that individual element has been initialized, as per the first item. C programmers may be unaware that Pascal allows each individual array to be non-zero-based (e.g., *weight*: **array** [1957..2020] **of** *pounds*), but this doesn't add any significant complication.

Additionally, Pascal is somewhat unusual in that it allows the programmer to specify that a scalar variable will store only a certain range of values. C has somewhat similar functionality with "bitfields" in records, where each integer field can be specified as taking a given number of bits, but Pascal completely generalizes this to ranges, exactly like array subscripts; for instance, *year_of_interest*: 1957..2020 declares a variable that may contain values only in the given range. As TEX and METAFONT make liberal use of this feature, it's also worth checking at runtime that each assignment to such a variable is within its declared range of valid values.

Similarly, if a procedure has a formal parameter with a limited range, at runtime each actual parameter of each call should be checked for validity. Checking that returned values from functions also match their signatures rounds out the list of range checks. It's also worth checking for overflow on each addition, subtraction, and multiplication operation (leaving the hardware to watch for division by zero).

3 Toolchain background and an edge case: empty change files

To do this checking, I wrote a purpose-built transpiler that inputs Knuthian Pascal and outputs very vanilla C code that implements all these runtime checks. The transpiler is about 7,600 lines of C and the runtime library about 2,500 lines.

TEX and METAFONT come with their own test input files, trip.tex and trap.mf, with instructions on how to use each to verify ports. These test files are designed to execute every line of code (with minor exceptions such as fatal error messages), and thus provide one good way to try out the transpiler output. Of course, generating the code involves running Tangle on all of the above WEB files, which tests it; and doing the complete trip and trap test regimens runs the rest of the programs; except for Weave, which I tested separately on all the above WEB files as well. Finally, I also ran a number of large documents that require only original (traditional? ur? un-enhanced? Knuthian?) TEX.

All of the Pascal files are generated directly from Tangle using empty change files, so everything is tested exactly as in DEK's master sources, unmodified, with two exceptions: First, *ready_already*, as mentioned above, gets specially tagged as being "unmemchecked", so it won't trigger a failure when it's read before being written. Second, the line of "dirty Pascal" code in the `tex.web` module ⟨Display the value of *glue_set*(*p*)⟩ that tries to detect invalid floating-point values that a bug may have caused to be stored in a floating point field, has been removed, as this attempt itself would be detected as trying to access the wrong variant (discussed below). Of course, as this later module is indexed under "system dependencies" and "dirty Pascal" in DEK's sources, this is an expected spot for a platform-specific change.

And one bug was found, manifested in both Tangle and Weave: they share a bunch of code that deals with text file reading, which is where the bug appeared. The module ⟨Read from *web_file* and maybe turn on *changing*⟩ contains this code:

```
...
else if $limit = change\_limit$ then
  if $buffer[0] = change\_buffer[0]$ then
    if $change\_limit > 0$ then $check\_change$;
...
```

to see if the current line from the WEB file matches the first line after an @x in the change file. First it tests that the line lengths match, then that the first characters match, and only then does it go to all the expense of actually calling a function (*check_change*) that sees if the lines fully match. (Computers used to be slow, so one could argue that this was a reasonable optimization, rather than paying the price of just calling *check_change* every time.) The bug occurs when the change file is completely empty, in which case *change_buffer*[0] is uninitialized when this code tries to read it. It's interesting to note that I would not have bumped into this bug if not for the fact that the transpiler handles DEK's code directly, and thus most of my change files are in fact empty.

However, rather to my dismay, in normal, non-undefined-behavior-checking-mode, no matter what junk might happen to be in *change_buffer*[0], Tangle and Weave still operate properly (since *change_limit* will be zero in this case, so *check_change* won't be called anyway). So no one will ever be affected by this bug, leaving the philosophical question as to whether it's actually a bug or not. A redeeming aspect is that to fix this quasi-bug, one can simply remove the middle line of the code shown! (Since, happily, *lines_dont_match* ultimately checks the first character as appropriate anyway.) This may be the first bug I've ever encountered where simply removing code fixes it. In fact, the **if** $limit = change\_limit$ **then** check can be removed, too!

A few cases of fetching uninitialized variables were also detected in METAFONT. But in all these cases, the value fetched doesn't ever get looked at as METAFONT continues to execute. For the record: The procedure *recycle_value* starts with

```
if $t < dependent$ then $v := value(p)$
```

and then uses v in some of the subsequent cases in its switch statement. It turns out that in some of the cases where v is not used, $value(p)$ hadn't ever been assigned to. Note that the existing code tries to avoid the fetch when it's not going to use the value, but the condition $t<dependent$ isn't correct. The other cases are in *copy_path*, its mirror *htap_ypoc*, and *scan_expression*, all of which copy *right*/*left* x/y values, some of which may not have been previously written to (when the source record came from, say, *new_knot*). Again, these copied, uninitialized values are not subsequently used.

4 Variant records and homegrown memory management

Another potential case of "undefined behavior" concerns "variant records" (known as "unions" to C programmers). Every time a member field of a union is read from, we must check that the most recent write to that variable was to that same member. This is very important for TeX and METAFONT, as they make extensive use of variant records; see particularly the definition of *memory_word*.

TeX and METAFONT never allocate dynamic memory or deal with pointers, so we needn't worry about checking pointer dereference validity. But they do their own form of memory management within the big *mem* array; see *get_node* and *free_node* for how a linked-list of free blocks is maintained. Note that this whole scheme uses indexes into the *mem* array as "pointers", so they can be stored in 2 bytes. (Or, for larger capacity TeXs, 4 bytes, which used to make it harder to explain why "real" pointers weren't used, but recently many platforms have switched to using 8-byte pointers exclusively, so TeX's 4-byte pseudo-pointers are again a space-saver.)

Of course, this approach to memory management is not typical. In addition to pointers being smaller, TeX implements a zero-overhead allocation scheme: as compared with most implementations of *malloc* in C libraries, where each allocated item takes 8 or more extra bytes of storage beyond what the caller requested, in TeX there are no extra bytes per allocation. Recall that TeX was developed on a machine with a data address space of only about half a megabyte, so it's a tight squeeze to fit in an entire macro package, hyphenation rules, font metrics, etc., while leaving enough room to store a whole page's worth of boxes and glue, etc. Every byte counted, and it's fair to say that things are packed to the gills (this can also be seen in Weave, which makes two passes over the input file rather than try to fit everything into memory).

As mentioned above, the mechanism for catching uninitialized variables involves keeping an extra bit of information per variable to indicate if it's "readable" or not. It's fairly straightforward to add another extra bit to control variables' "writable" status. Then, by augmenting the code in *get_node* and *free_node* to make special calls that turn the "writable" bits on and off, respectively, for the entire node being allocated or freed, access-after-free errors get caught automatically. Additionally, we can arrange by using a huge *mem* array that no freed slot is ever reused later as part a subsequent allocation, thereby ensuring that there's no chance of an illicit read ever getting lucky and going undetected. This requires about 200 lines in TeX's change file, including more specific safeties, such as setting the static glue specs *zero_glue*, *fil_glue*, …, *fil_neg_glue* to be not writable (and all tests of TeX were run with and without this change, just to be sure that no bugs get hidden by it; ditto for subsequent changes mentioned herein).

The entire suite of programs pass all these checks, as one might expect, given their robustness in the field. Also, if I recall correctly, in the 1980s there was an especially good Pascal compiler from DEC for their VAX/VMS systems that was able to detect these sorts of errors, and someone in the TeX user community reported a few bugs of this sort that it found.

But there's yet another, more subtle, type of problem left to consider. The issue with keeping track of which member of a union is "active" has already been mentioned. But TeX goes a step further, and re-uses members for different purposes in different contexts. For instance, the *in_state_record* is how TeX keeps track of the current line of input from a file, with member fields that tell where the line begins and ends, and where the next character to read is within those limits. But when the current input is instead from a macro, these same fields get used to now keep track of the start of the macro's token list, and where along the way the next token is to be fetched from, etc. Some of the fields are given new names with a simple `WEB` macro that redirects to the old name; other fields just get reused with the same name (such as *start*, which is a fine name to indicate either the start of a token list or start of a line, even though in one case it's a pointer into *mem*, and in the other an index into the input buffer of characters). But either way, how do we make sure that a value stored with one meaning isn't attempted to be interpreted with the other meaning?

The answer is to manually introduce new member fields in separate variants to distinguish the two contexts, thus reducing the problem to one that's already been addressed. This takes some manual labor to examine every use of each symbol, and assign it to one of the two variants, but it's not too onerous, resulting in fewer than 200 lines in the change file.

Quite a bit more extreme are the *memory_words* in *mem*; they have more than three dozen different possible interpretations: *height*, *width*, *depth*, *glue_stretch*, *glue_shrink*, *penalty*, etc., etc. The interesting twist here is that multiple node types share various fields under a common name and offset; both boxes and rules have *width*, *height*, and *depth*, but only a box has a *shift_amount*; and a kern also has a *width*, but no *height* or *depth*. The trick here is to put each of these field types into its own variant in *memory_word*, so they get checked individually. So the *width* of a box is the same sort of thing as the *width* of a rule, but different than the other sorts of things that other node types have at offset 1.

While separating out all the different uses of *memory_word*, we get an additional opportunity for checking ranges. For example, consider TeX's "delimiter fields", which hold family and character values. These normally get stored in byte-sized fields in a *memory_word* record, but the *fam* is always supposed to be in the range 0..15. So, the newly introduced variant can actually specify that its *fam* field is of this range type, with the result that all the checking logic will get kicked off by the transpiler. TeX has other similar cases, including some where the permissible values are more like an enumeration, and are in fact turned into one; for instance the *stretch_order* and *shrink_order* fields of a glue specification take values of the enumerated type *glue_ord*.

Quite a bit of manual effort was involved with this set of alterations, requiring over 1100 lines in TeX's change file, but with satisfactory results.

5 An edgier case: unbalanced braces at end of file

And, again, all this additional checking finds a bug, this time in TeX. It's in the use of the input state (discussed above). If the module ⟨Input the next line of *read_file*[*m*]⟩ encounters an end-of-file at a time when braces haven't been balanced, a call to the *error* reporting routine is made. But this happens before the formalities of setting up the input state to properly represent the input line have happened. So, an uninitialized field of the *in_state_record* gets read, leading to the failure.

In fact, the Trip test hits this very situation (not surprisingly, since it attempts to hit every line of code in TeX, including each error message). It was never noticed because of a happy coincidence in which the junk values happen to produce a reasonable result, and TeX continues uninjured. Results could be more disastrous with non-Trip inputs. The same bug can occur with the fatal error "`*** (cannot \read from terminal in nonstop modes)`" occurs, but that's even more of an edge case, and isn't tested for (as all fatal errors are not).

☡☡ In particular, ⟨Input and store tokens from the next line of the file⟩ hasn't yet fallen through to the code that sets *loc* and *limit* when the *error* call happens. When *error* does ⟨Pseudoprint the line⟩, we're in trouble, since it thinks that *loc* and *limit* tell where the contents of the problematic line are.

☡☡ When a non-checking TeX gets to line 415 of trip.tex, where the "File ended within `\read`" case is tested, it kind of lucks out: *limit* happens to be a left-over zero (from a left-over *param_start*, no doubt), while *loc* is a big number (similarly representing a left-over token location), and *start* is actually a correct pointer into *buffer*. So ⟨Pseudoprint the line⟩ randomly checks *buffer*[0] for *end_line_char*, then in any case skips its for loop, and happily shows the two empty context lines (lines 6167–68 in `trip.log`):

```
l.6167 <read 0>
l.6168
```

To see this bug in action, we can create a file `unbal.tex` containing a single "`{`" character, and create a second file `readbug.tex` containing:

```
\catcode`{=1 \catcode`}=2 \catcode`#=6
\openin1 unbal
\def\A#1#2#3#4#5#6#7#8#9{\read1to \x}
\def\B#1#2#3#4#5#6#7#8#9{\A#1#2#3#4#5#6#7#8#9 \relax}
\def\C#1#2#3#4#5#6#7#8#9{\B#1#2#3#4#5#6#7#8#9 \relax}
\def\D#1#2#3#4#5#6#7#8#9{\C#1#2#3#4#5#6#7#8#9 \relax}
\def\E#1#2#3#4#5#6#7#8#9{\D#1#2#3#4#5#6#7#8#9 \relax}
\E123456789
```

Then, running `virtex readbug` results in a nonsense `<read 1>` context:

```
! File ended within \read.
<read 1> {^^M#5#6#7#8#9{\D#
```

The trick here is that all those parameters cause *param_start* to be 36, which then gets used as a bogus value for *limit* when *show_context* is called, resulting in unrelated stuff in *buffer* being shown as the context.

6 More stress: checking constants

Thus far, all of the testing has used as input only files which are part of the basic TeX distribution as it came from Stanford. So, Tangle and Weave get tested with all of the WEB sources; TeX and METAFONT have their Trip and Trap test files, but TeX also runs all the Weave output, as well as *The TeXbook* and *The METAFONTbook*, while METAFONT runs all of Computer Modern at various resolutions, and so on.

This represents a lot of stress, but doesn't hit everything. One additional direction I have tried pushing was to see that the ⟨Check the "constant" values for consistency⟩ checks were complete and accurate. Nothing of much importance showed up, other than there being no check that *buf_size* is at least big enough to hold the longest built-in primitive name (which happens to be a tie, at 21 characters each, between "abovedisplayshortskip" and "belowdisplayshortskip").

Finally, the only deeply-embedded constant in TeX that can't be changed at all (as far as I have noticed, anyway) is that the hyphenation routines only work on words of length up to 64. Or is it 63? Or is it only on the first 64 letters of a word? And does that mean only 63 possible hyphenation points, or would it include a possible hyphen after the 64th letter of a longer word? And where is this specified in *The TeXbook*? Trying to figure this out by reading the code is a challenge, as there are various 63's, 64's, and 65's scattered about (the latter having to do with adding sentinels to the word being hyphenated so that the pattern matching has something to match for beginning- and ending-of-word; and/or adding a byte that indicates the "current language" when looking up hyphenation exceptions).

So, I created a test file containing the four lines (abridged here):

```
\lefthyphenmin=0 \righthyphenmin=0
\hyphenation{-a-b-...-y-z-a-b-...-y-z-a-b-...-y-z}
\showhyphens{ab...yzab...yzab...yz}
\end
```

and tried it out. Sure enough, a bug occurs: the code tries to store a value that's not in the declared range of the receiving variable. In particular, in the *hyphenate* function, when ⟨Look for the word *hc*[1..*hn*] in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found⟩ is called, *hn* can already be 63, but then this module increments it to 64 for a while (to fit the *cur_lang* byte), which puts *hn* out of range for a *small_number*.

On common architectures, this bug probably won't actually change TeX's behavior, since *hn* will no doubt be stored in a full byte, which means it will be able to actually store the value 64 properly. By the way, I'd guess this bug got introduced when multiple-language support was added to the hyphenation code; the arrays grew by one to be able to append the language byte, but the declaration of *hn* got overlooked (easy to do, as it didn't show an explicit 0..63 range).

7 Need more `\input`

A big limitation in all this is the small number of plain TeX and METAFONT files I've been able to use as test input. The issue is that this is absolutely plain, original TeX and METAFONT, as created by DEK, without any of the added features of pdfTeX, MetaPost, etc. So, I'm on the lookout for TeX documents that use only macro packages that work on unmodified TeX. Any help in this regard would be appreciated, and I'm happy to share credit for finding any bugs that your devious macros or lengthy tome might turn up!

◇ David Fuchs
`plain-tex-tests (at) tug dot org`
`https://tug.org/texmfbug/`