Converting TEX from WEB to cweb

Martin Ruckert

Why translate TEX from WEB to cweb?

A long term goal brought me to construct the program `web2w` that translates TEX from `WEB` to cweb: I plan to derive from the TEX sources a new kind of TEX that is influenced by the means and necessities of current software and hardware.

The major change in that new kind of TEX will be the separation of the TEX frontend: the processing of `.tex` files, from the TEX backend: the rendering of paragraphs and pages.

Let's look, for example, at ebooks: Current ebooks are of rather modest typographic quality. Just compiling TEX documents to a standard ebook format, for example epub, does not work because a lot of information that is used by TEX to produce good looking pages is not available in these formats. So I need to cut TEX in two pieces: a frontend that reads TEX input, and a backend that renders pixels on a page. The frontend will not know about the final page size because the size of the output medium may change while we read—for example by turning a mobile device from landscape to portrait mode. On the other hand, the computational resources of the backend are usually limited because a mobile device has a limited supply of electrical energy. So we should do as much as we can in the frontend and postpone only what needs to be postponed to the backend. In between front and back, we need a nice new file format that is compact and efficient and transports the necessary information between both parts.

For the work described above, I will need to work with the TEX source code and make substantial changes. The common tool chain from the TEX Live project uses `tangle` to convert `tex.web` into Pascal code (`tex.pas`) which is then translated by `web2c` into C code. In the course of this process also other features of a modern TEX distribution are added. Hence the translation process is not just a syntactic transformation but also introduces semantic changes. So it seemed not the best solution for my project. Instead, I wanted to have cweb [7] source code for TEX, which I could modify and translate to C simply by running `ctangle`.

The result of my conversion effort was surprisingly good, so I decided to make it available on `ctan.org` [10] and to present it here, in the hope others may find it useful when tinkering with TEX.

How the program web2w was written

On December 9, 2016, I started to implement web2w with the overall goal to generate a tex.w file that is as close as possible to the tex.web input file, and can be used to produce tex.tex and tex.c simply by running the standard tools ctangle and cweave.

web2w was not written following an established software engineering workflow as we teach it in our software engineering classes. Instead the development was driven by an ongoing exploration of the problem at hand where the daily dose of success or failure would determine the direction I would go on the next day.

This description of my program development approach sounds a bit like "rapid prototyping". But "prototype" implies the future existence of a "final version" and I do not intend to produce such a "final version". Actually I have no intention to finish the prototype either, and I might change it in the future in unpredictable ways. Instead I have documented the development process as a literate program [6]. So in terms of literature, this is not an epic novel with a carefully designed plot, but more like the diary of an explorer who sets out to travel through yet uncharted territories.

The territory ahead of me was the program TEX written by Donald E. Knuth using the WEB language [4] as a literate program. As such, it contains snippets of code in the programming language Pascal — Pascal-H to be precise. Pascal-H is Charles Hedrick's modification of a compiler for the DEC-system-10 that was originally developed at the University of Hamburg (cf. [1], see [5]). So I could not expect to find a pure "Standard Pascal". But then the implementation of TEX deliberately does not use the full set of features that the Pascal language offers. Hence at the beginning, it was unclear to me what problems I would encounter with the subset of Pascal that is actually used in TEX.

Further, the problem was not the translation of Pascal to C. A program that does this is available as part of the TEX Live project: web2c [11] translates the Pascal code that is produced using tangle from tex.web into C code. The C code that is generated this way cannot, however, be regarded as human readable source. The following example might illustrate this: figure 1 shows the WEB code for the function *new_null_box*. The result of translating it to C by web2c can be seen in figure 3. In contrast, figure 2 shows what web2w will achieve.

web2c has desugared the sweet code written by Knuth to make it unpalatable to human beings; the only use you can make of it is feeding it to a C compiler. In contrast, web2w tries to create source code that is as close to the original as possible but still translates Pascal to C. For example, see the last statement in the *new_null_box* function: where C has a **return** statement, Pascal assigns the return value to the function name. A simple translation, sufficient for a C compiler, can just replace the function name by "Result" (an identifier that is not used in the implementation of TEX) and add "return Result;" at the end of the function (see figure 3). A translation that strives to produce nice code should, however, avoid such ugly code.

136. The *new_null_box* function returns a pointer to an *hlist_node* in which all subfields have the values corresponding to '\hbox{}'. The *subtype* field is set to *min_quarterword*, since that's the desired *span_count* value if this *hlist_node* is changed to an *unset_node*.

```
function new_null_box: pointer;
          { creates a new box node }
  var p: pointer;   { the new node }
  begin p ← get_node(box_node_size);
  type(p) ← hlist_node; subtype(p) ← min_quarterword;
  width(p) ← 0; depth(p) ← 0; height(p) ← 0;
  shift_amount(p) ← 0; list_ptr(p) ← null;
  glue_sign(p) ← normal; glue_order(p) ← normal;
  set_glue_ratio_zero(glue_set(p)); new_null_box ← p;
  end;
```

Fig. 1: WEB code of *new_null_box*

136. The *new_null_box* function returns a pointer to an *hlist_node* in which all subfields have the values corresponding to '\hbox{}'. The *subtype* field is set to *min_quarterword*, since that's the desired *span_count* value if this *hlist_node* is changed to an *unset_node*.

```
pointer new_null_box(void)
  /* creates a new box node */
{ pointer p;     /* the new node */
  p = get_node(box_node_size); type(p) = hlist_node;
  subtype(p) = min_quarterword; width(p) = 0;
  depth(p) = 0; height(p) = 0; shift_amount(p) = 0;
  list_ptr(p) = null; glue_sign(p) = normal;
  glue_order(p) = normal;
  set_glue_ratio_zero(glue_set(p)); return p;
}
```

Fig. 2: cweb code of *new_null_box*

The structure of web2w

The program web2w works in three phases: First I run the input file tex.web through a scanner producing tokens. The pattern matching is done using flex. During scanning, information about macros,

```
halfword
newnullbox ( void )
{
  register halfword Result; newnullbox_regmem
  halfword p  ;
  p = getnode ( 7 ) ;
  mem [p ].hh.b0 = 0 ;
  mem [p ].hh.b1 = 0 ;
  mem [p + 1 ].cint = 0 ;
  mem [p + 2 ].cint = 0 ;
  mem [p + 3 ].cint = 0 ;
  mem [p + 4 ].cint = 0 ;
  mem [p + 5 ].hh .v.RH = -268435455L ;
  mem [p + 5 ].hh.b0 = 0 ;
  mem [p + 5 ].hh.b1 = 0 ;
  mem [p + 6 ].gr = 0.0 ;
  Result = p ;
  return Result ;
}
```

Fig. 3: `web2c` code of *new_null_box*

identifiers, and modules is gathered and stored. The tokens then form a doubly linked list, so that later I can traverse the source file forward and backward. Further, every token has a `link` field which is used to connect related tokens. For example, I link an opening parenthesis to the matching closing parenthesis, and the start of a comment to the end of the comment.

After scanning comes parsing. The parser is generated using `bison` from a modified Pascal grammar [3]. To run the parser, I feed it with tokens, rearranged to the order that `tangle` would produce, expanding macros and modules as I go. While parsing, I gather information about the Pascal code. At the beginning, I tended to use this information immediately to rearrange the token sequence just parsed. Later, I learned the hard way (modules that were modified on the first encounter would later be fed to the parser in the modified form) that it is better to leave the token sequence untouched and just annotate it with information needed to transform it in the next stage.

A technique that proved to be very useful is connecting the key tokens of a Pascal structure using the `link` field. For example, connecting the "**case**" token with its "**do**" token makes it easy to print the expression that is between these tokens without knowing anything about its actual structure and placing it between "**switch** (" and ")".

The final stage is the generation of cweb output. Here the token sequence is traversed again in input file order. This time the traversal will stop at the warning signs put up during the first two passes,

```
function new_character(f : internal_font_number;
        c : eight_bits): pointer;
  label exit;
  var p: pointer;  {newly allocated node}
  begin if font_bc[f] ≤ c then
    if font_ec[f] ≥ c then
      if char_exists(char_info(f)(qi(c))) then
        begin p ← get_avail; font(p) ← f;
        character(p) ← qi(c); new_character ← p;
        return;
        end;
  char_warning(f, c); new_character ← null;
exit: end;
```

Fig. 4: `WEB` code of *new_character*

```
pointer new_character(internal_font_number
        f, eight_bits c)
{ pointer p;     /* newly allocated node */
  if (font_bc[f] ≤ c)
    if (font_ec[f] ≥ c)
      if (char_exists(char_info(f)(qi(c)))) {
        p = get_avail( ); font(p) = f;
        character(p) = qi(c); return p;
      }
  char_warning(f, c); return null;
}
```

Fig. 5: cweb code of *new_character*

use the information gathered so far, and rewrite the token sequence as gently and respectfully as possible from Pascal to C.

Et voilà! `tex.w` is ready — well, almost. I have to apply a last patch file, for instance to adapt documentation relying on `webmac.tex` so that it will work with `cwebmac.tex`, and make changes that do not deserve a more general treatment. The final file is then called `ctex.w` from which I obtain `ctex.c` and `ctex.tex` merely by applying `ctangle` and `cweave`. Using "`cc ctex.c -lm -o ctex`", I get a running `ctex` that passes the trip test.

Major challenges

I have already illustrated the different treatment of function return values in Pascal and C with figures 1 and 2. Of course, replacing "*new_null_box* ← *p*;" by "**return** *p*;" is a valid transformation only if the assignment is in a tail position. A tail position is a position where the control flow directly leads to the end of the function body as illustrated by figure 4 and 5. It is possible to detect tail positions by traversing the Pascal parse tree constructed during phase 2.

The **return** statement inside the **if** in figure 5 is correct because the Pascal code in figure 4 contains

a "**return**". This "**return**", however, is a macro defined as "**goto** *exit*", and "*exit*" is a numeric macro defined as "10". In C, "**return**" is a reserved word and "*exit*" is a function in the C standard library, so something has to be done. Fortunately, if all goto-statements that lead to a given label can be eliminated, as is the case in figure 5, the label can be eliminated as well. So you see no "*exit*:" preceding the final "}".

Another seemingly small problem is the different use of semicolons in C and Pascal. While in C a semicolon follows an expression to make it into a statement, in Pascal the semicolon connects two statements into a statement sequence. For example, if an assignment precedes an "**else**", in Pascal you write "`x:=0 else`" whereas in C you write "`x=0; else`"; but no additional semicolon is needed if a compound statement precedes the "**else**". When converting `tex.web`, a total of 1119 semicolons need to be inserted at the right places. Speaking of the right place: Consider the following `WEB` code:

```
if s ≥ str_ptr then s ← "???"
    { this can't happen }
else if s < 256 then
```

Where should the semicolon go? Directly preceding the "**else**"? Probably not! I should insert the semicolon after the last token of the assignment. But this turns out to be rather difficult: assignments can be spread over several macros or modules which can be used multiple times; so the right place to insert a semicolon in one instance can be the wrong place in another instance. `web2w` starts at the **else**, searches backward, skips the comment and the newlines, and then places the semicolon like this:

```
if (s ≥ str_ptr) s = ⟨ "???" 1381 ⟩;
    /* this can't happen */
else if (s < 256)
```

But look at what happened to the string `"???"`. Strings enclosed in C-like double quotes receive a special treatment by `tangle`: the strings are collected in a string pool file and are replaced by string numbers in the Pascal output. No such mechanism is available in `ctangle`. My first attempt was to replace the string handling of TEX and keep the C-like strings in the source code. The string pool is, however, hardwired into the program and it is used not only for static strings but also for strings created at runtime, for example to hold the names of control sequences. So I tried a hybrid approach: keeping strings that are used only for output (error messages for example) and translating other strings to string numbers using the module expansion mechanism of `ctangle`, like this:

```
1381.
#define str_256 "???"
⟨ "???" 1381 ⟩ ≡ 256
This code is used in section 59.
```

I generate for each string a module, that will expand to the correct number, here 256; and I define a macro *str_256* that I use to initialize the string pool variables.

In retrospect, seeing how nicely this method works, I ponder if I should have decided to avoid the hybrid approach and used modules for all strings. It would have reduced the number of changes to the source file considerably.

Another major difference between Pascal and C is the use of subrange types. In Pascal subrange types are used to specify the range of valid indices when defining arrays. While most arrays used in TEX start with index zero, not all do. In the first case they can be implemented as C arrays which always start at index zero; in the latter case, I define a zero based array having the right size, and add a "0" to the name. Then I define a constant pointer initialized by the address of the zero based array plus/minus a suitable offset so that I can use this pointer as a replacement for the Pascal array.

When subrange types are used to define variables, I replace subrange types by the next largest C standard integer type as defined in `stdint.h` — which works most of the time. But consider the code

```
var p: 0 .. nest_size;   { index into nest }
  ⋮
for p ← nest_ptr downto 0 do
```

where *nest_size* = 40. Translating this to

```
uint8_t p;     /* index into nest */
  ⋮
for (p = nest_ptr; p ≥ 0; p−−)
```

would result in an infinite loop because p would never become less than zero; instead it would wrap around. So in this (and 21 similar cases), variables used in **for** loops must be declared to be of type **int**.

Related work

As described by Taco Hoekwater in "LuaTEX says goodbye to Pascal" [2], the source code of TEX was rewritten as a part of LuaTEX project as a collection of cweb files. This conversion proceeded in two steps: first `TEX.WEB` was converted into separate plain C files while keeping the comments; at a much later date, those separate files were converted back

```
341. Now we're ready to take the plunge into get_next itself. Parts of this routine are executed more often than any other instructions of TEX.
  define switch = 25  { a label in get_next }
  define start_cs = 26  { another }
procedure get_next;
        { sets cur_cmd, cur_chr, cur_cs to next token }
  label restart,  { go here to get the next input token }
    switch,
        { go here to eat the next character from a file }
    reswitch,  { go here to digest it again }
    start_cs,  { go here to start looking for a control
        sequence }
    found,  { go here when a control sequence has been
        found }
    exit;  { go here when the next input token has
        been got }
  var k: 0 .. buf_size;  { an index into buffer }
    t: halfword;  { a token }
    cat: 0 .. max_char_code;
        { cat_code(cur_chr), usually }
    c, cc: ASCII_code;
        { constituents of a possible expanded code }
    d: 2 .. 3;  { number of excess characters in an
        expanded code }
  begin restart: cur_cs ← 0;
  if state ≠ token_list then ⟨ Input from external file,
        goto restart if no input found 343 ⟩
  else ⟨ Input from token list, goto restart if end of list
        or if a parameter needs to be expanded 357 ⟩;
  ⟨ If an alignment entry has just ended, take
      appropriate action 342 ⟩;
exit: end;
```

Fig. 6: WEB code of *get_next*

to cweb format. In this process, not only the specific extensions of the LuaTEX project were added, but TEX was also enhanced by adding features using the ε-TEX, pdfTEX, and Aleph/Omega change files. These extensions are required for LATEX and modern, convenient TEX distributions. The conversion was done manually except for a few global regular expression replacements. I have included three versions of the *get_next* function to illustrate the differences between the traditional `TEX.WEB` by Don Knuth (Fig. 6), my code (Fig. 7), the code found as part of LuaTEX (Fig. 8).

One can see that `web2w` has eliminated the label declarations, but left the comments in the code. Certainly this is something that could be improved in a later version of `web2w`, by moving such comments to the line where the label is defined in C.

```
341. Now we're ready to take the plunge into get_next itself. Parts of this routine are executed more often than any other instructions of TEX.
void get_next(void)
  /* sets cur_cmd, cur_chr, cur_cs to next token */
{   /* go here to get the next input token */
    /* go here to eat the next character from a file */
    /* go here to digest it again */   /* go here to
      start looking for a control sequence */
    /* go here when a control sequence has been
      found */   /* go here when the next input
      token has been got */
  uint16_t k;   /* an index into buffer */
  halfword t;   /* a token */
  uint8_t cat;   /* cat_code(cur_chr), usually */
  ASCII_code c, cc;
    /* constituents of a possible expanded code */
  uint8_t d;   /* number of excess characters in an
      expanded code */
restart: cur_cs = 0;
  if (state ≠ token_list) ⟨ Input from external file,
      goto restart if no input found 343 ⟩
  else ⟨ Input from token list, goto restart if
      end of list or if a parameter needs to be
      expanded 357 ⟩;
  ⟨ If an alignment entry has just ended, take
      appropriate action 342 ⟩;
}
```

Fig. 7: `web2w` code of *get_next*

In the LuaTEX version, these comments have disappeared together with the labels, while the comment that follows after the procedure header was converted into the text of a new section.

`web2w` retained the definitions of local variables, converting the subrange types to the closest possible type from `stdint.h`. For example, "*k*: 0 .. *buf_size*" was converted to "**uint16_t** *k*". *buf_size* is defined earlier in `ctex.w` as *buf_size* = 500. Note that changing this to *buf_size* = 70000 would not force a corresponding change to **uint32_t** in the definition of *k*. Only changing the definition in `TEX.WEB` and rerunning `web2w` would propagate this change. This is an inherent difficulty of the translation from Pascal to C. In the LuaTEX version, the local variables have disappeared and were moved to the subroutines called by *get_next*.

The module references present in the original `WEB` code (namely ⟨Input from external file ...⟩, ⟨Input from token list ...⟩, ⟨If an alignment entry ...⟩), are not retained in the LuaTEX version. Instead, LuaTEX converts them either to function calls or expands the modules turning the module name

34. Now we're ready to take the plunge into *get_next* itself. Parts of this routine are executed more often than any other instructions of TEX.

35. sets *cur_cmd*, *cur_chr*, *cur_cs* to next token

```
void get_next(void)
{
RESTART: cur_cs = 0;
  if (istate ≠ token_list) {      /* Input from external
        file, goto restart if no input found */
    if (¬get_next_file()) goto RESTART;
  }
  else {
    if (iloc ≡ null) {
      end_token_list();
      goto RESTART;
        /* list exhausted, resume previous level */
    }
    else if (¬get_next_tokenlist()) {
      goto RESTART;
        /* parameter needs to be expanded */
    }
  }     /* If an alignment entry has just ended, take
        appropriate action */
  if ((cur_cmd ≡ tab_mark_cmd ∨ cur_cmd ≡
        car_ret_cmd) ∧ align_state ≡ 0) {
    insert_vj_template();
    goto RESTART;
  }
}
```

Fig. 8: LuaTEX code of *get_next*

into a comment. Part of the problem of turning modules into subroutines is the translation of the **goto** *restart* statements without creating non-local **goto**s. LuaTEX solves the problem by using boolean functions that tell the calling routine through their return values whether a **goto** *restart* is called for. In contrast, the automatic translation by `web2w` stays close to the original code, avoiding this problem by retaining the modules.

Conclusion

Using the `web2w` program, the TEX source code can be converted to the cweb language, designed for the generation of C code and pretty documentation. The resulting code is very close to the original code by Knuth; its readability is surprisingly good. While manual translation is considerably more work, it offers the possibility (and temptation) of changing the code more drastically. Automatic translation can be achieved with limited effort but is less flexible and its result is by necessity closer to the original code.

The `web2w.w` program itself and the converted TEX source code, `ctex.w`, are available on CTAN for download [8, 10]. Since `web2w` is a literate program, you can also buy it as a book [9].

References

[1] C. O. Grosse-Lindemann and H. H. Nagel. Postlude to a PASCAL-compiler bootstrap on a DECsystem-10. *Software: Practice and Experience*, 6(1):29–42, 1976.

[2] Taco Hoekwater. LuaTEX says goodbye to Pascal. *TUGboat*, 30(3):136–140, 2009. `https://tug.org/TUGboat/tb30-3/tb96hoekwater-pascal.pdf`.

[3] Kathleen Jensen and Niklaus Wirth. *PASCAL: User Manual and Report*. Springer Verlag, New York, 1975.

[4] Donald E. Knuth. *The WEB system of structured documentation*. Stanford University, Computer Science Dept., Stanford, CA, 1983. STAN-CS-83-980. `https://ctan.org/pkg/cweb`.

[5] Donald E. Knuth. *TEX: The Program*. Computers & Typesetting, Volume B. Addison-Wesley, 1986.

[6] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[7] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison Wesley, 1994. `https://ctan.org/pkg/cweb`.

[8] Martin Ruckert. `ctex.w`*: A TEX implementation*. `http://mirrors.ctan.org/web/web2w/ctex.w`, 2017.

[9] Martin Ruckert. *WEB to cweb*. CreateSpace, 2017. ISBN 1-548-58234-4. `https://amazon.com/dp/1548582344`.

[10] Martin Ruckert. `web2w`*: Converting TEX from WEB to cweb*. `https://ctan.org/pkg/web2w`, 2017.

[11] *Web2C: A TEX implementation*. `https://tug.org/web2c`.

◇ Martin Ruckert
Hochschule München
Lothstrasse 64
80336 München
Germany
`ruckert (at) cs dot hm dot edu`