An HTML/CSS Schema for TEX Primitives – Generating High-Quality Responsive HTML from generic TEX

Dennis Müller

This paper uses sTEX3. The semantically annotated XHTML version of this paper is available at url.mathhub.info/tug23css

Abstract

I present a schema for translating TEX primitives to HTML/CSS. This translation can serve as a basis for (very) low-level TEX-to-HTML converters, and is in fact used by the RusTEX system – a (somewhat experimental) implementation of a TEX engine in Rust, used to convert LATEX documents to XHTML– for that purpose.

Notably, the schema is accurate enough to yield surprisingly decent (and surprisingly often “the exactly right”) results on surprisingly many “high-level” LATEX macros, which makes it adequate to use in lieu of (and often even instead of) dedicated support for macros and packages.

1 Introduction

Translating LATEX documents (partially or fully) to HTML is a difficult problem, primarily because the two document formats address very different needs: TEX is intended to produce statically layout documents with fixed dimensions, ultimately representing ink on paper. HTML on the other hand assumes a variety of differently sized and scaled screens and consequently prefers to express layouts in more abstract terms, the typesetting of which are ultimately left to the browser to interpret; ideally responsively – i.e. we want the document layout to adapt to different screen sizes, ranging from 8K desktop monitors to cell phone screens.

This means that there is no one “correct” way to convert TEX to HTML– rather there is a plurality of choices to be made; most notably, which aspects of the static layout with fixed dimensions described by TEX code to preserve, or discard in favour of leaving them up to the rendering engine, explaining the plurality of existing converters.

Naturally, many LATEX macros are somewhat aligned with tags in HTML; for example, sectioning macros (like `\chapter`, `\section`, etc.) correspond to `<h1>`, `<h2>`, etc; and the `\begin{itemize}` and `\begin{enumerate}` environments and the `\item` macro correspond to `<ul>`, `<ol>` and `<li>`, respectively. Most converters therefore opt for the reasonable strategy of mapping common LATEX macros directly to their closest HTML relatives, with no or minimal usage of (simple) CSS; effectively focusing on preserving the *document semantics* of the used constructs (e.g. “paragraph”, “section heading”, “unordered list”). In many situations, this is the natural approach to pursue, especially if we can reasonably assume that the document sources to be converted are sufficiently “uniform”, so that we can provide a similarly uniform CSS style sheet to style them, and this is largely the way existent converters work. To name just a few:

LATEXML [6] focuses strongly on the *semantics*, using XML as the primary output format and heuristically determining an author’s intended semantics of everything from text paragraphs (definitions, examples, theorems, etc.) down to the meaning of individual symbols in mathematical formulae; achieving great success with `ar5iv.org`, hosting HTML documents generated from TEX sources available on `arxiv.org`. TEX4ht [12] focuses on plain HTML as output with minimal styling, going as far as to replace the `\LaTeX` macro by the plain ASCII string “`LaTeX`”. Pandoc [10] largely focuses on the most important macros and environments with analogues in all of its supported document format to convert between any two of them, e.g. TEX, Markdown, HTML, or `docx`. Mathjax [5] focuses exclusively on macros for mathematical formulae and symbols, allowing to use TEX syntax in HTML documents directly, which are subsequently replaced via JavaScript by the intended presentation.

However, the approach described above has notable drawbacks: Firstly, it requires special treatment of LATEX macros that plain TEX would expand into primitives instead, and the amount of LATEX macros is virtually unlimited – CTAN has (currently) a collection of 6399 packages, tendency growing, which get updated regularly, and authors can add their own macros at any point. Supporting only the former is a neverending task, and providing direct HTML translations for the latter is impossible. This is made worse by the very real and ubiquitous practice among LATEX users of copy-pasting and reusing various macro definitions and preambles assembled from stackoverflow, friends and colleagues, and handed down for (by now *literally*) generations, even in situations where (unbeknownst to them) “official” packages with better solutions (possibly supported by HTML converters) exist.

For example, I myself have happily reused the following macro definition for years:

```
\usepackage{amsmath,amssymb}
\def\forkindep{\mathrel{\raise0.2ex\hbox
  {\ooalign{\hidewidth$\vert$\hidewidth
```

```
    \cr\raise-0.9ex\hbox{$\smile$}}}}}
```

...neither knowing nor caring what it actually does other than that it allowed me to typeset $A \forksnot_C B$ ("A and B are *forking-independent* (or *non-forking*) over C"; a concept in model theory)[1] – despite there existing a unicode symbol (0x2ADD) and a corresponding LaTeX macro `\forksnot` in the `unicode-math` package. If we want to maximise coverage, we therefore need a reasonable strategy for arbitrarily elaborate unexpected LaTeX macros.

Secondly, by generating rather plain HTML, we guarantee that the resulting presentation is *neutral* and can be easily adapted by users via their own CSS stylesheets – the "morally correct" thing to do. However, it also severely clashes with the expectations of (casual) users that the result looks roughly the same as the PDF does. After all, the way LaTeX documents are written by authors is optimized for a particular layout and arrangement of document elements. Subsequently discarding them in favor of as-plain-as-possible HTML that optimizes more for the "document semantics" of the components than their (precise) optics yields plain looking HTML that is immediately perceived as ugly, "not what I want" and requires lots of massaging to achieve a similar aesthetic level as the PDF generated by pdflatex does. And *aesthetics matter* – that's why TeX was built in the first place.

Thirdly, by focusing on supporting as many LaTeX macros as possible directly, conversion engines tend to neglect support for primitives in multiple senses of "support" – indeed, I found it difficult finding any existing TeX documents of mine that "survive" any of the existing HTML converters for a realistic comparison, typically dying with no output or only initial, badly formatted fragments.

The RusTeX system is a TeX-to-HTML converter born out of our needs in the sTeX project [4, 8]. The `stex` package allows for annotating LaTeX documents (in particular mathematical formulae and statements) with their (flexi-)formal semantics. These documents are subsequently converted to HTML, preserving both the (informal) document layouts as well as the semantic annotations in such a way, that knowledge management services acting on the semantics can be subsequently integrated via JavaScript. Our existing corpora of sTeX documents cover a wide range, from individual fragments (definitions, theorem statements, remarks,...) up to research papers, lecture slides in `beamer`, and book-like lecture notes that usually *include* the slides between text fragments, all of them using a multitude of (typical and untypical, official and custom) packages, preambles and stylings.

We consequently want to translate the sources for all these heterogeneous documents to HTML such that 1. the results look as similar to their PDF counterparts as possible, 2. the semantic annotations are preserved as XML attributes, and 3. (most importantly) conversion succeeds for any error-free document, regardless of packages and macros used, so that at least the semantic annotations can be extracted, even if the presentation is occasionally (somewhat) broken.

Contribution Motivated by the above, this paper describes RusTeX's rather extremal point in the design space of LaTeX-to-HTML converters: The goal is to mimic the core TeX expansion mechanism (i.e. pdflatex) as closely as possible and map the resulting sequence of TeX primitives to (primarily) `<div>`s with CSS attributes, while avoiding the neverending amount of work required for the special treatment for non-primitive TeX macros. Ideally, this allows for achieving full error-free coverage with respect to converting full documents, and yielding HTML that looks reasonably close to what a user would expect.

Of course, if we *only* care about aesthetics, we might as well render the generated PDF in the browser directly. So as an addendum to the above, we should add the desideratum that the HTML remain "reasonably recognizable as HTML": for example, plain text in paragraphs (or horizontal boxes) should actually be represented as plain text in the resulting HTML– in fact, as much as possible we want to leave to the browser what a browser does best: break lines in paragraphs, size boxes based on their contents (where we want them to be), and arrange components based on available (screen) space, according to constraints imposed by our CSS schema.

RusTeX's git repository [11] contains a `.tex`-file with test cases for (and beyond) all the following, and the HTML generated from them for direct comparison. Additionally it contains the PDF and HTML produced from my Ph.D. dissertation [7], which serves as a particularly good test case for several reasons:

1. I was a typical LaTeX user when I wrote it, with no particular knowledge of TeX's internal workings, and hence unbiased by what I would nowadays do to avoid problems.

2. I spent a lot of effort on making it look nice by the usual means – copy-pasting from elsewhere

[1] Possibly sourced from `tex.stackexchange.com/questions/42093/what-is-the-latex-symbol-for-forking-independent-model-theory` – I needed and found it some time around 2013.

and using whatever package google tells me to use to achieve the desired effect.

3. It is a 215 page document using everything from elaborate formulas, syntax-highlighted code listings, various figures and tables, and color-coded environments (using `tcolorboxes`) for remarks, theorems, examples, definitions, etc.

The HTML generated from our sTeX corpora can be found at `url.mathhub.info/stex`, including this paper (see link above), which thus additionally serves as a demonstration of the examples below (notably, with two column mode deactivated). They also power our *course portal* at `courses.voll-ki.fau.de`, where students at our university can access semantically annotated course materials and various didactic services generated from them.

The full CSS schema can be found at [2].

Disclaimer Note that I am not arguing to eschew dedicated support for LaTeX and package macros entirely – document semantics can be important, for example for accessibility reasons. Additionally, while the translation presented here is surprisingly effective, it has clear limitations, especially on the scale of *individual characters* (see section 9).

Hence, the contents of this paper should be seen as a reasonable *fallback* strategy usable *in conjunction with* dedicated support for macros. Indeed, RusTeX too currently implements (few) package macros as well, namely `\url`, `\not` and `\cancel`, `\underbrace` and `\overbrace`, `\marginpar`, `\begin{wrapfig}`, and (somewhat embarrassingly) `\LaTeX`.

In fact, if this paper has a purpose beyond reporting on what I consider to be an interesting experiment, it should be the following: *Taking TeX primitives seriously pays off aesthetically*, can spare a lot of work and effort, and where possible, I encourage developers of TeX-to-HTML converters to take them seriously *in addition to* dedicated support for macros.

Furthermore, many of the techniques described below are the result of more-or-less informed experimentation; in many cases, better ways to represent TeX primitives in HTML might exist. I appreciate feedback and suggestions for improvements.

2 General Architecture

As mentioned, RusTeX attempts to mimic the behaviour of `pdflatex` as closely as possible. As such, it implements the behaviour of the primitive commands available in plain TeX, eTeX and pdfTeX, amounting to 293 + 47 commands, excluding primitive “register-like” commands such as `\everyhbox`, `\baselineskip` or `\linepenalty`. Their precise behaviour has been determined from (obviously) the bible [3] and the manuals for eTeX and pdfTeX, but also often reverse engineered via extensive experimentation.

At the start of the program, a user’s `pdftexconfig.tex` and `latex.ltx` are located using `kpsewhich` and processed first. This entails that a user needs to have a LaTeX distribution set up, but subsequently makes sure that RusTeX behaves as close to the local LaTeX setup as possible.

Tokens are expanded in the expected manner down to the primitives, which cause state changes, impact expansion, or ultimately end up fully processed in RusTeX’s *stomach* waiting to be output as HTML. The latter primitives are the subject of this paper.

`pgf` (and thus `tikz`) is handled via an adapted version of the existing SVG driver and thus omitted here. Images are inserted directly in the HTML in Base64-encoding.

In lieu of a *shipout routine*, box registers for *floats* (as well as `\insert`s such as footnotes) are occasionally heuristically inspected and inserted, but this mechanism is due for a more adequate treatment and hence also omitted.

2.1 Trees and Fonts

Naturally, HTML is a tree structure of nested nodes. Somewhat counterintuitively, so are TeX’s stomach elements, but unfortunately at the cost of attaching information such as the current font, font size, color, etc. directly to the individual “character boxes”. If we wanted to introduce a `<span>` node for every individual character, we could mimic this directly in HTML– however, this approach is too extreme even for my taste. Luckily, in almost every situation where colors and fonts are changed, the changes are achieved via LaTeX macros that align with TeX’s ‘stomach tree”. For example,

```
\textbf{\textcolor{blue}{some} \emph{text}}
```

clearly entails a tree of font and color changes, which ideally should be represented as a corresponding HTML tree:

```
<span style="font-weight:bold">
  <span style="text-color:blue">some</span>
  <span style="font-style:italic">text</span>
</span>
```

And indeed, all three macros (`\textbf`, `\textcolor`, `\emph`) introduce TeX groups for their arguments, assuring that these changes too reflect a tree structure.

[2] `github.com/slatex/RusTeX/blob/master/rustex/src/resources/html.css`

Consequently, RusTEX can (somewhat) safely add special nodes to the stomach on font changes, changes to the color stack, or *links* (as produced by `\pdfstartlink`). As these are (usually) local to the current TEX group, the stomach consequently also keeps track of when TEX groups are opened and closed. If such changes (i.e. their start and end points) conflict with other stomach element's delimiters (such as boxes or paragraphs), they are appropriately closed and subsequently reopened, e.g.:

```
Some paragraph \begingroup \itshape
this is italic \par
New paragraph, still italic \endgroup not
italic anymore
```

> Some paragraph *this is italic*
>
> *New paragraph, still italic* not italic anymore

would yield HTML similar to:

```
<div class="paragraph">
  Some paragraph
  <span style="font-style:italic">
    this is italic
  </span>
</div>
<div class="paragraph">
  <span style="font-style:italic">
    New paragraph, still italic
  </span>
  not italic anymore
</div>
```

In general, the nodes produced by font changes and similar commands are considered "*annotations*": If these nodes have no children, or a single child that modifies the same CSS property, they are discarded or replaced by their only child. If they have a single child or are the only child of their parent node, the corresponding `style`-attribute is attached to the relevant node directly. Only in the remaining case is an actual `<span>` node produced in the output HTML.

To deal with fonts in general, it should be noted that most TEX fonts are freely available in a web-compatible format (e.g. `otf`) online; we *could* consequently use the actual fonts used by TEX in the output PDF. In practice, we prefer to have adequate Unicode characters in the HTML output, rather than ASCII characters representing a position in a font table. Consequently, RusTEX instead hardcodes fonts as pairs of 1. a map from ASCII codes to Unicode strings and, 2. a sequence of font modifiers (e.g. *bold*, *italic*). The former is used to produce actual characters, the latter to choose appropriate CSS attributes as above.

Currently, RusTEX fixes Latin Modern as the font family used, but somewhat nonsensically obtains font metrics the same way as TEX, by processing the `tfm`-files on demand [2], providing only rough approximations of the actual values (in HTML).

2.2 Global Document Setup

At `\begin{document}`, RusTEX determines 1. the current font and its size, 2. the page width (as determined by `\pdfpagewidth`) and 3. the text width (as determined by `\hsize`), and attaches them as corresponding CSS attributes to the `<body>` node – the page width determining the `max-width` and $(\langle\textit{page width}\rangle - \langle\textit{text width}\rangle)/2$ determining the `padding-left` and `padding-right` properties. The latter is important to accomodate e.g. `\marginpar` and related mechanisms, and is discussed more precisely in section 5.

3 Boxes and Dimensions

Clearly, the most important primitives to get "right" are (horizontal or vertical) *boxes*, produced by `\hbox`, `\vbox` and variants (`\vtop`, `\vcenter`), as they are the primary means that more elaborate macros use to achieve their aims. They also serve as a good example of the complexities involved when translating to HTML.

Boxes have five important numerical values that matter with respect to how they are typeset: `width`, `height`, `depth`, `spread` and `to`, which we will discuss shortly.

Horizontal boxes (as produced by `\hbox`) – as the name suggests – have their contents arranged horizontally, and vertical ones vertically. This is nicely analogous to the CSS *flex model*, so naturally, we can associate boxes with CSS flex display values. An entire document can be thought of as a single top-level vertical box. Hence:

```
.hbox, .vbox, .body {
  display: inline-flex;
}
.vbox, .body { flex-direction: column; }
.hbox { flex-direction: row; }
```

An important distinction that matters here is that between the actual *contents* of the box and its *boundary*. Usually, the dimensions of a box are computed from the dimensions of its children – which, conveniently, is analogous to HTML/CSS, so in the typical case we do not need to bother with them at all and leave those up to the rendering engine:

```
.hbox, .vbox, .body {
  width: min-content;
  height: min-content;
}
```

Whenever possible, we *avoid* precisely assigning dimensional values in HTML and defer to the ones computed by the rendering engine. This is important to account for discrepancies between HTML and TEX, e.g. regarding the precise heights of characters, lines, paragraphs , etc.

However, the dimensions of a box can be changed after the fact, using the `\wd`, `\ht` and `\dp` commands (corresponding to `width`, `height` and `depth`, respectively). If these dimensions are changed, the *contents* and how they are layed out are not changed at all, but the typesetting algorithm, when putting “ink to paper”, will proceed *as if* the box had the provided dimensions. This allows macros to layer boxes *on top* of each other; in the (very common) most extreme case by making boxes take up no space at all. For example:

```
\setbox\myregister\hbox{some content}
\wd\myregister=0pt \ht\myregister=0pt
\dp\myregister=0pt
\box\myregister other content
```

This will produce a horizontal box with the content “some content” with all dimensions being 0 from the point of view of the output algorithm, meaning the “other content” following the box will be put directly *on top* of the box, like so:

```
some content
other content
```

Hence, we *do* have to occasionally consider the actual (computed or assigned) dimensions of TEX boxes and other elements.

Regarding boxes, we attach actual values for `width`/`height` to their HTML nodes *if and only if* they have been *assigned* fixed values, and let

```
.hbox, .vbox { overflow: visible; }
```

We can then achieve the same effect in HTML via:

```
<div class="hbox" style="width:0;height:0;">
  some content
</div> other content
```

3.1 `width`/`height` vs. `to`

Things get more interesting if the assigned values for the dimensions of the box are *larger* than the actual box contents – this tells us how we need to align the contents of boxes vertically and horizontally. This, however, is also where the `to`-value of a box comes into play:

Setting (exemplary) `\wd`=⟨*val*⟩ for a horizontal box, as mentioned, does not actually impact the way the box *content* is layed out. `\hbox to`=⟨*to-val*⟩`{...}` however *does*, while also setting the `width` of the box: The `to`-attribute instructs TEX to arrange the contents of the box “in line with” the box being ⟨*to-val*⟩ wide,e.g.:

```
\hbox{some box content}
\hbox to \textwidth{some box content}
```

```
some box content
some                  box                  content
```

This example is deceptive in that it suggests the box contents were evenly spread out across the ⟨*to-val*⟩ of the box, but this is not so. Consider:

```
\hbox to \textwidth{
  \hbox{some}\hbox{box}\hbox{content}
}
\hbox to \textwidth{%
  \hbox{some}\hbox{box}\hbox{content}%
}
```

```
                 someboxcontent
someboxcontent
```

It’s not that the individual content elements in the box are spread out evenly; instead, they are left-aligned and *space characters* (and newline characters, which are treated like spaces) behave (roughly) as if followed by `\hfil` – i.e. they take up as much space as they can in the containing `\hbox`. And while subsequently, the box has a width of ⟨*to-val*⟩, changing that with `\wd` is possible:

```
\setbox\myregister\hbox to \textwidth{%
  \hbox{some}\hbox{box}\hbox{content}%
}\wd\myregister=0pt \box\myregister
\setbox\myregister\hbox to \textwidth{%
  some box content%
}\wd\myregister=0pt \box\myregister
```

```
someboxcontent     box                  content
```

This distinction between the three values `width`, `to`, and “total width of the box’s children” forces us to actually distinguish between a) the box itself (i.e. its contents) with its (potential) `to` value, and b) its “boundary box”, i.e. subsequently assigned `width`s and `height`s. The same holds analogously for the `to` value and `height` of a vertical box:

```
.hbox { text-align: left; }
.vbox { justify-content: flex-start; }
.hbox-container, .vbox-container {}
```

where the `.hbox-container`-class is used for *assigned* `width`s and `height`s, and `to` translates to the width of the `.hbox` itself. Making spaces behave as they should in an \hbox forces us to style them accordingly:

```
.space-in-hbox {
  display: inline-block;
  margin-left: auto;
  margin-right: auto;
}
```

Using this class for spaces (directly) in \hboxes makes the remaining content stretched across the full width of the box, as in the examples above.

Notably, TEX allows for *negative* values in dimensions, which CSS does not. To capture the resulting behaviour, whenever a dimension (exemplary `width`) is < 0, we set the `width` CSS property to 0, and attach (in this case) `margin-right:`⟨*width*⟩ to the HTML node (analogously `margin-top` for `height`).

Finally, the `spread` parameter can be used *instead* of `to` and *adds* the provided dimension to the computed width/height of the box; e.g. if `\hbox{foo}` has width 15pt, then `\hbox spread 15pt{foo}` has width $15 + 15 = 30$pt:

Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque.foo   Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque.

Annoyingly, the only way to accomodate this seems to be to compute the “original” value, add the `spread` value, and attach that as the final width/height to the `<div>` node.

3.2 Depth and Rules

So far, we have only considered `width` and `height`, but TEX has an additional dimension for boxes that CSS does not: `depth`, which measures the extent to which a given box extends *below* the baseline of the parent box. Depth is rarely important, or rather, matters primarily when manipulating individual characters, which CSS is currently not capable of for reasons explained later. However, notable not uncommon exceptions are explicitly *assigned* depth values, in particular for \vtop boxes.

To better understand depth, we should turn our attention the the \vrule primitive, which produces a colored box of the provided dimensions:[3]

```
Lorem ...
\vrule width 10pt height 10pt depth 10pt
Lorem ...
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque.█Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque.

This creates a black box with 10pt width and a total 20pt height, centered at the *baseline* of the current line: extending 10pt *above* the baseline (the `height`) and 10pt *below* (the `depth`).[4]

Such a box with the right dimensions can be easily produced using CSS:

```
.vrule {
  display: inline-block;
}
```

The individual `<div>`s are then provided `background`, `width` and `height` (=`height`+`depth`) properties corresponding to the color and the dimensions of the `\vrule` – in the above example[5]

```
style="background:#000000;height:20pt"
```

The tricky part is ensuring that the box is correctly positioned with respect to the surrounding text (or other elements). As above, the solution is to wrap the `.vrule` `<div>` in a `.vrule-container` with the same height as the inner `<div>`, and adding `margin-bottom:-`⟨*depth*⟩ to the inner `.vrule`. This not only allows for moving the box the specified amount below the baseline, but also makes sure that the “boundary” that the rendering engine computes for positioning elements has the relevant dimensions as well.

If a rule has no explicitly provided width/height, it is computed by TEX to be 0.6pt wide, and a length fitting the current box:

```
\hbox{ \vrule
 \vbox{ \hbox{some} \hrule \hbox{text}}
\vrule }
```

| some
text |

We can easily set the width of the `\hrule` with `width:100%` to achieve the same effect. Unfortunately, the same does not work with `\vrule` and its height in HTML, as an artifact of when and how the heights of boxes are computed by the rendering engine. In those situations, we have to distinguish between paragraphs and `\hbox`es: In the former case we heuristically set the height to the current font size, in the latter (since we are in a flex box), we can set

[3] `\hrule` is implemented analogously, except for using `display:block` instead of `inline-block`.

[4] Note the gap between the second and third line of text, caused by the depth ob the `\vrule`.

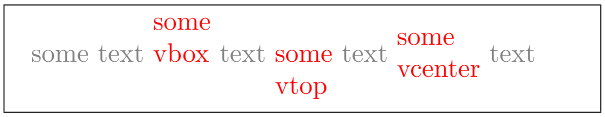
[5] For simplicity’s sake, we will use the same dimensions (in `pt`) in both TEX code and CSS; in actual practice, we scale `1pt` in TEX to some value in `px` units.

align-self:stretch to make the rule fit the containing box.

3.3 \vbox vs \vtop vs \vcenter

\vtop behaves like \vbox, except that where a \vbox is vertically aligned at the *bottom* of the parent box's baseline, a \vtop is vertically aligned at the top with the surrounding text, extending downwards. \vcenter is vertically aligned at the center and is only allowed in math mode:

```
some text \vbox{\hbox{some}\hbox{vbox}}
text \vtop{\hbox{some}\hbox{vtop}}
text $\vcenter{\hbox{some}\hbox{vcenter}}$
text
```



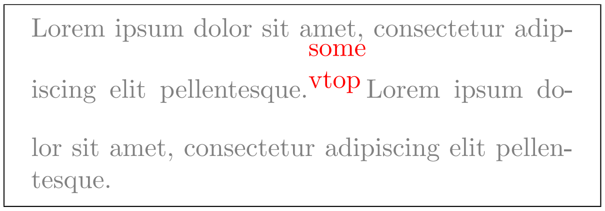
some
some text vbox text some text some text vcenter text
vtop

Internally, the three types of vertical boxes differ precisely in their a priori depths and heights. As long as these are not subsequently reassigned (using \ht and \dp), we can achieve the same effect much more accurately by using the vertical-align property, that covers the same primary *intent* of the three types of vertical boxes:

```
.vbox{ vertical-align: bottom }
.vtop{ vertical-align: baseline }
.vcenter{ vertical-align: middle }
```

We now need to be careful with changing the *height* of a \vtop box, however: Since the primary vertical dimension of a \vtop corresponds to its *depth* (below the baseline), *increasing* its height actually corresponds to moving the box contents upwards *without changing the amount of space* it takes up *below* the baseline:[6]

```
Lorem ...
\setbox\myregister\vtop{\hbox{some}\hbox{vtop}}
\ht\myregister=20pt\box\myregister
Lorem ...
```



Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque. some vtop Lorem ipsum dolor sit amet, consectetur adipiscing elit pellentesque.

This can be approximated in HTML by setting both the margin-top *and* bottom CSS properties of the .vbox-container to the value ⟨*height*⟩−⟨*current line height*⟩: The bottom property moves the box upwards, while the margin-top property makes sure that the boundary box grows acordingly, instead of the moved box overlapping with other elements.[7]

Conversely, if we manipulate the *depth* of a \vtop, we can set the height of the .vtop HTML node itself to ⟨*depth*⟩+⟨*current line height*⟩.

Annoyingly, it now turns out that height/depth manipulations on \vboxes and \vtops (respectively) do not play well with vertical-align CSS properties within paragraphs – the boxes are not correctly aligned vertically. When explicitly setting these dimensions, it is therefore necessary to, as with \vrule, introduce an intermediate HTML node with class .vbox-height-container to achieve the effect.

4 Paragraphs

At a first glance, paragraphs in TEX seem largely stright-forward:

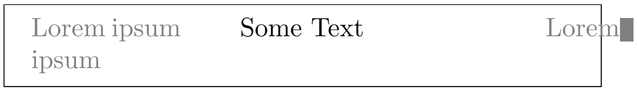
```
.paragraph {
  text-align:justify;
  display: inline-block;
  margin-top: auto;
}
```

The margin-top:auto assures that paragraphs are vertically aligned at the bottom of \vboxes.

Any horizontal material (text, \noindent, \unhbox,...) outside of a paragraph or an \hbox (and similar constructions) *opens* a new paragraph, and \par closes it again.

If we were primarily interested in document semantics without caring about the page layout dictated by TEX, we could be done at this point. However, in TEX, paragraphs have fixed widths dictated by several parameters and commands, including \hsize, \leftskip, \rightskip, \hangindent and \hangafter, and \parshape. This matters when a paragraph is opened inside a \vbox. Consider e.g.

```
Lorem ipsum \vbox{Some Text} Lorem ipsum
```



Lorem ipsum Some Text Lorem ipsum

The Some Text in the \vbox opens a new paragraph, including indentation, and that paragraph has width \hsize, regardless of its contents. The \vbox itself then inherits the full width of the containing paragraph.[8]

[6] Again, note how the three lines in the paragraph are pushed apart by the unchanged depth and new height of the box

[7] The same idea is used for \raise/\lower.

[8] Here, we set \hsize to a smaller value to attempt to demonstrate the effect without breaking the layouting of this very document too much.

Approximating this behaviour (in the absence of dedicated macro support) matters, for example to accomodate `\begin{minipage}`s, `tcolorbox` and similar packages. This is also one instance where TeX is significantly more flexible than HTML/CSS: `\hangindent` and `\parshape` do not have CSS equivalents. While in principle in might be possible to "emulate" them using empty `<div>` nodes with `float` attributes, we currently ignore them and proceed as if the whole paragraph were typeset according to the rules applying to the last line; e.g. the last entry in the `\parshape` list.

The relevant parameters can subsequently be condensed into three attributes, in the simplest case computed thusly: 1. the actual width of the text (`\hspace`−(`\leftskip`+`\rightskip`)), and 2. left and right margins (`\leftskip` and `\rightskip`), which we translate to the CSS attributes `min-width`, `margin-left` and `margin-right`, respectively.

Notably, to accomodate macros that make use of computed dimensions of various boxes, we need to approximate TeX's line breaking algorithm to make sure that the computed heights of paragraphs are reasonably accurate.

5 Responsiveness and Relative Widths

The above suggests, that we need to hardcode the absolute widths of both the document as a whole (in the sense of `\textwidth`/`\pagewidth`) as well as the widths or paragraphs and `\hbox`es. This is of course undesirable in that it destroys responsive layout in HTML. Ideally, we would prefer to use *relative* widths in terms of percentages.

Regarding the document width, this is easily resolved: Instead of letting `width:`⟨*text width*⟩, we set `max-width:`⟨*text width*⟩. This way, the page accomodates smaller screen, but if enough screen space is available will default to the size the document was originally designed for.

Relative widths in general however only work as expected if the direct parent of a node has a fixed assigned width, and as previously mentioned, in as far as possible we want to defer the precise dimensions of HTML nodes to the rendering engine. Moreover, once we have a box width `width:0`, no percentage will get us back to a non-zero value. Both problems were solvable if CSS would allow for inheriting attribute values from arbitrary ancestors, but since it does not, we need to be more creative:

Instead of directly inheriting, we can use a *custom* CSS property `--current-width` and initialize it as `--current-width:min(100vw,`⟨*text width*⟩`);` `width:var(--current-width)` in the body. This achieves the same effect as the more naive approach above, but now allows for stating other widths in the body of the HTML node as values relative to the `--current-width` attribute.

Using this approach, all relative widths in a document are now relative to the *current document's* initial `\textwidth`. This is problematic in the context of sTeX, where the `\inputref` macro largely replaces TeX's `\input`: Besides allowing for referencing source files relative to a math archive (i.e. a "library" of document snippets), which is important for building modular libraries, when converting to HTML `\inputref` simply inserts a reference to the file, that can subsequently be dynamically inserted into the referencing document. This obviates the need to both reprocess the same file for every context in which it occurs, as well as to rebuild all referencing files every time any of the `\inputref`ed files change. Notably, such `\inputref`s often occur deeply nested, e.g. a file with a short individual definition might be `\inputref`ed in an `\begin{itemize}` environment in a definition block in a framed beamer slide within lecture notes. This entails that we would like to inherit widths from *the closest ancestor with a fixed assigned width* > 0 (e.g. the innermost `\item` in the example above) rather than the `<body>`, and update the value of `--current-width` accordingly, to accomodate any document context in which the HTML node might (dynamically) occur.

Hence, when encountering e.g. a (top-level) `\vbox` with width `0.5\textwidth` (e.g. a `\begin{minipage}`), we would like to do:

```
<div class="vbox" style="--current-width:calc(
  0.5 * var(--current-width));
  width:var(--current-width)">...</div>
```

Unfortunately, CSS does not allow for self-referential attribute updates; so we have to use an intermediary custom attribute `--temp-width` and an inner `<span>` to do the following:

```
<div class="vbox" style="--temp-width:calc(
    0.5 * var(--current-width));
    width:var(--temp-width)">
  <span style="display:contents;
    --current-width:var(--temp-width)
  ">...</span></div>
```

to achieve the desired effect. While this is ugly from an implementation point of view, it allows for variable viewport widths and solves the problem with inheriting widths *through* boxes of size 0.

6 Skips and Text Alignment

In section 4, we acted as if `\leftskip` and `\rightskip` where simple dimensions – i.e. values of unit `pt`.

Skips actually have three components: A base dimension, an (optional) *stretch* factor, and an (optional) *shrink* factor. A skip represents a (horizontal or vertical) space that is *ideally* ⟨*base dimension*⟩ wide/high, but can stretch or shrink according to the other two components to fit the current page layout. Stretch and shrink factors have one of four units `pt`, `fil`, `fill` or `filll`, the latter three representing "increasingly infinite" stretch/shrink factors.

Skips are used to introduce vertical or horizontal space, using the `\hskip` and `\vskip` commands. Focusing solely on their base dimensions for now, both can be represented as empty `<div>` nodes with corresponding `margin-left` or `margin-bottom` values, respectively. Conveniently, this works with both positive and negative base dimensions, and we can use the same mechanism for `\kern`, which for all practical purposes behaves like `\hskip` or `\vskip` with zero stretch/shrink. This allows us to cover both of the following cases:

```
\noindent some text \hskip20pt some text\par
\noindent some text \hskip-20pt some text
```

```
some text       some text
some textsome text
```

If we add a strech factor, we can e.g. achieve the following:

```
\noindent some text \hskip20pt plus 1filll
some text\par
```

```
some text                                Some text
```

Unfortunately, CSS has no analogue for stretch and shrink factors. For *shrink*, this largely causes no serious issues. *Stretch* factors however are primarily used to achieve (primarily horizontal) *alignment*. Left-aligned, centered, or right-aligned content is achieved in TEX by inserting corresponding skips; so the best we can do is to represent skips as the CSS `text-align` property:

If `\leftskip` or `\rightskip` have stretch factors, we compare them and set the alignment for the paragraph accordingly. For `\hbox`, we need to inspect the contents of the box for initial and terminal occurrences of relevant skips, compare them, and derive the intended alignment depending on which is "bigger".

Additionally, we can add `margin-left:auto` to the `<div>`s corresponding to skips iff they have a stretch factor of (at least) `1fil`; however, this only works in `\hbox`es (not in paragraphs), and does not necessarily behave right in conjunction with other skips. Thankfully, text alignment seems to be the primary regularly occuring situation where skips are noticable and important to represent accurately in the HTML, which this heuristical approach seems to cover reasonably well – while discrepancies between PDF and HTML can be easily found, they are usually not severe.

7 Math Mode

For stomach elements in math mode, we naturally use Presentation MathML. Translating the relevant primitives to MathML is largely straight forward and covered elsewhere [9], with the slight "modernization" that we prefer CSS over MathML attributes. Since the font used for MathML depends on the rendering engine, and some of them are rather unsatisfactory (e.g. vanilla Firefox under Ubuntu), we can explicitly set the font to `Latin Modern Math` for a more unifying look. Skips and kerns are implemented as above, but using `<mspace>` nodes instead of `<div>`.

Regarding font sizes, we can either defer to the rendering engine or leave that up to TEX– in which case we need to make sure that we override the CSS rules imposed by the rendering engine via:

```
msub > :nth-child(2), msup > :nth-child(2),
mfrac > * , mover > :not(:first-child),
munder > :not(:first-child) {font-size:inherit}
```

More pressingly however, occurrences of `\hbox` or `\vbox` in math mode require us to "escape" back to HTML in `<math>` elements. While not officially supported, using `<mtext>` nodes for that works well in both Firefox and Chromium (and with some hacking with MathJax). However, when doing so, various CSS properties are inherited from those set by the default stylesheet for MathML. Hence, whenever we escape back to horizontal or vertical mode, we explicitly insert the parameters of the current text font, and set:

```
mtext {
  letter-spacing: initial;
  word-spacing: initial;
  display: inline-flex;
}
```

As mentioned in [9], spacing around operators (i.e. `<mo>` nodes) is governed by an operator dictionary. The spacing rules are in principle well-chosen and best left to the rendering engine. TEX can change these however, using the commands `\mathop`, `\mathbin`, etc.

To accomodate this functionality, we can explicitly set left and right padding based on TEX's math character class, and set:

```
mo {padding-left: 0;padding-right: 0}
```

Notably, this works (as of May 2023) in Firefox, but not in Chromium-based browsers[9], where the spacing determined by the operator dictionaries is effectively a *minimum* that can not be reduced further.

Changing these spacing factors can occasionally be important when composing symbols from more primitive ones. For example, the `\Longrightarrow` macro $\Longrightarrow$ concatenates the symbols $=$ and $\Rightarrow$ with a negative `\kern` between them - in which case unintended spacing between the two symbols can break the intended result.

8 \halign

The `\halign` command is the primary means LaTeX packages use to layout *tables*, and not surprisingly, its closest correspondants in HTML are `<table>` nodes. However, as with text alignment, effects that in HTML are achieved via attributes of the parent node (`<table>`, `<tr>` or `<td>`) are achieved in TeX via content elements *in* the individual cells – or between them: Where a table in HTML is exactly a sequence of rows consisting of cells, in TeX, the `\noalign` command allows for inserting vertical material *between* rows, which is used to insert horizontal lines (e.g. `\hline`) or determine the spacing between rows. Borders and spacing between cells are achieved via `\vrule`s and skips.

Hence, we have to face two major problems when translating `\halign`s to `<table>`s:

1. If we want to accomodate spacing, text alignments and borders, we need to "parse" the contents of cells and `\noalign` blocks to determine which CSS attributes to attach to the `<table>`, `<tr>` and `<td>` nodes. This is worsened by the fact that the margin attributes on `<td>` and `<tr>` nodes have no actual effect.
2. The *height* of a `<tr>` is computed from the *actual* height of its children, and even enclosing a whole cell in a `<div>` with `height:0` does not change the actual height of the relevant `<tr>`.

While the former problem is inconvenient but solvable, the latter becomes severe if we consider less obvious situations that `\halign` is used for: For example, the `\forkindep` macro mentioned above (i.e. $⫝$) uses `\ooalign` to combine the two characters $|$ and $\smile$, which uses an `\halign` to superimpose them, forcing us to make the rows narrower than `<tr>`s allow for.

Therefore we use the CSS grid model for `\halign` rather than the (seemingly more adequate) `<table>`:

```
.halign {
  display:inline-grid;
  width: fit-content;
  grid-auto-rows: auto;
}
```

with cells being styled like `.hbox` with the additional attributes `height:100%;width:100%`, and any `\halign` with n columns being given the additional CSS attribute `grid-template-columns:repeat(n,1fr)`. This aligns the individual cells almost exactly like `<table>` would, but gives us the more control over their intended heights. `\noalign` vertical material can now be inserted in a `.vbox` `<div>` with `grid-column:span n`. Notably, this entirely obviates the need to implement special rules for visible borders or spacing between rows/columns: The existing treatment for `\vrule`/`\hrule` and skips produces (almost universally) the desired output out of the box.

Notably, empty cells in `\halign` are not actually empty. Consider:

```
\halign{#&#\cr a&b\cr c&d\cr&\cr e&f\cr}
```

```
ab
cd

ef
```

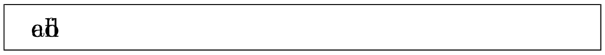
Note that the third row really is entirely empty, with no spacing involved. Instead, we get a row that has roughly the same height as the other three. We can remedy this effect via:

```
\baselineskip=0pt\relax
\halign{#&#\cr a&b\cr c&d\cr&\cr e&f\cr}
```

```
ab
cd
ef
```

or do even more ridiculous things:

```
\baselineskip=0pt\relax
\lineskiplimit=-100pt\relax
\halign{#&#\cr a&b\cr c&d\cr&\cr e&f\cr}
```



This entails, that we now need to take `\baselineskip` and `\lineskiplimit` into account and use them to compute `min-height` (for `\baselineskip`) or `height` values (in case of sufficiently negative `\lineskiplimit` values) for the cell's HTML node.

9 Limitations

This brings us to the first insurmountable difference between TeX and CSS: *lines*. A line of text in

[9] conversely, scaling brackets properly with `stretchy="true"` seems to not work in Firefox as of yet.

TEX consists of individual character boxes with individual heights, widths and depths, and the spacing between lines is governed by the three parameters `\baselineskip` (the "default" distance between two baselines), `\lineskiplimit` (the minimally allowed distnce between the bottom of a line and the top of the subsequent one), and `\lineskip` (the minimal skip to insert between two lines, if their distance is below the `\lineskiplimit`). In particular, the height of a horizontal box containing e.g. a single character is entirely determined by the height of that particular character.

In contrast, a line of text in HTML/CSS has a *fixed* height of the current `line-height` value regardless of the occurring characters – and every single character counts as a "line": for every character, a *leading* space is inserted on top of it to make the containing box adhere to the `line-height`. This makes box manipulation on the level of individual characters currently (almost) impossible.[10]

One striking example for this is the `\LaTeX` macro, where the `A` is enclosed in a `\vbox`. RusTEX replaces its expansion by a simple `\raise\hbox` to achieve the (almost) same effect.

Situations where layouting critically depends on very precise positioning and sizing of boxes remains tricky. This is the case for example with the `tikzcd` package, where the nodes are layout as tables, with `pgf` arrows between the individual cells.

Various macros make use of LATEX floats in non-trivial ways, such as `\marginpar` and the `\begin{wrapfig` environment, making special treatment for them (as of yet) unavoidable.

The `xy` package is a clear example of where, due to its usage of custom fonts, there is currently no feasible way to achieve support in terms of TEX primitives alone; anecdotally, I have been told that a `pgf` driver for `xy` is in the works, which, if completed, would likely immediately work for RusTEX as well.

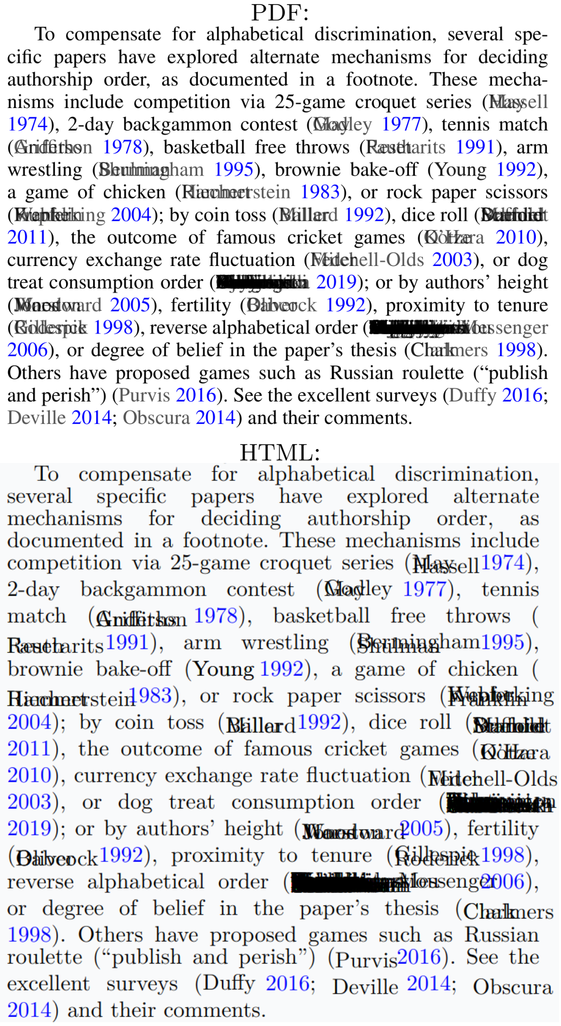
10 Conclusion

Despite the limitations mentioned above, the schema presented here works surprisingly well in a variety of cases. For example, list environments (`\begin{itemize}`, `\begin{enumerate}`, etc.), `\begin{lstlisting}`, figures, `\begin{algorithmic}`, `tcolorbox`, various environments for definitions, theorems and examples, `bibtex` and `biblatex`, and many other macros, environments and packages with often intricate options and configurations, work out of the box without special treatment and with the expected presentation in the HTML.

Indeed, it is certainly surprising how much can be achieved without providing dedicated implementations for non-primitive macros, to the point where I am nowadays more surprised if the schema *fails* than when it *succeeds*.

To mention one particular highlight: A tongue-in-cheek paper was published in May 2023 on `arxiv.org` that argued for solving the order-of-authors problem in scientific publishing by *overlaying all the author names on top of each other*, including instructions how to achieve that in both TEX and HTML [1].

Running RusTEX over the LATEX sources for the paper produced the right layout directly (see Figure 1).



PDF:

To compensate for alphabetical discrimination, several specific papers have explored alternate mechanisms for deciding authorship order, as documented in a footnote. These mechanisms include competition via 25-game croquet series (May Hassell 1974), 2-day backgammon contest (May Godley 1977), tennis match (Griffiths Anderson 1978), basketball free throws (Rautharits 1991), arm wrestling (Shulman Bermingham 1995), brownie bake-off (Young 1992), a game of chicken (Hammerstein 1983), or rock paper scissors (Kapferking 2004); by coin toss (Miller Ballard 1992), dice roll (Stafford 2011), the outcome of famous cricket games (Kotz O'Hara 2010), currency exchange rate fluctuation (Fitch Mitchell-Olds 2003), or dog treat consumption order ([illegible] 2019); or by authors' height (Woodward 2005), fertility (Babcock 1992), proximity to tenure (Roderick Gillespie 1998), reverse alphabetical order ([illegible] Messenger 2006), or degree of belief in the paper's thesis (Clark Chalmers 1998). Others have proposed games such as Russian roulette ("publish and perish") (Purvis 2016). See the excellent surveys (Duffy 2016; Deville 2014; Obscura 2014) and their comments.

HTML:

To compensate for alphabetical discrimination, several specific papers have explored alternate mechanisms for deciding authorship order, as documented in a footnote. These mechanisms include competition via 25-game croquet series (May Hassell1974), 2-day backgammon contest (May Godley 1977), tennis match (Griffiths Anderson 1978), basketball free throws (Rautharits1991), arm wrestling (Shulman Bermingham1995), brownie bake-off (Young 1992), a game of chicken (Hammerstein1983), or rock paper scissors (Kapferking 2004); by coin toss (Miller Ballard1992), dice roll (Stafford 2011), the outcome of famous cricket games (Kotz O'Hara 2010), currency exchange rate fluctuation (Fitch Mitchell-Olds 2003), or dog treat consumption order ([illegible] 2019); or by authors' height (Woodward2005), fertility (Babcock1992), proximity to tenure (Roderick Gillespie1998), reverse alphabetical order ([illegible] Messenger2006), or degree of belief in the paper's thesis (Clark Chalmers 1998). Others have proposed games such as Russian roulette ("publish and perish") (Purvis2016). See the excellent surveys (Duffy 2016; Deville 2014; Obscura 2014) and their comments.

Figure 1: Screenshots from [1] in PDF and RusTEX generated HTML

[10] A proposal to the W3C CSS WG regarding leading space, which would presumably help here, has been open since 2018: `github.com/w3c/csswg-drafts/issues/3240`

The most important aspect for generating adequate (and often great) HTML seems to be the "proper" treatment of `\hbox`/`\vbox`, `\hrule`/`\vrule` and skips/kerns, which RusTeX implements as described here. Their treatment should be relatively easy adaptable to, and usable by, other HTML converters as well, where "PDF-like" HTML output is desirable.

The most dire *limitations* are often related to intrinsic limitations of CSS– presumably, any extension of CSS that allows for more fine-grained control, especially on the character level, would allow for even better translations from TeX.

Future Work Naturally, some of the techniques described here have been slightly simplified and are augmented in practice via various heuristics that are still subject to experimentation and improvements. Other discrepancies or problems are usually addressed (if possible) as we become aware of them (which still happens regularly).

Acknowledgements The presented research is part of the VoLL-KI project, supported by the Bundesministerium für Bildung und Forschung (BMBF) under grant 16DHBKI089.

References

[1] Erik D. Demaine and Martin L. Demaine. *Every Author as First Author.* 2023. arXiv: `2304.01393 [cs.DL]`.

[2] David Fuchs. "TeX Font Metric files". In: *Communications of the TeX Users Group (TUGboat).* Vol. 2. 1. 1981, pp. 53–61. URL: `https://www.tug.org/TUGboat/tb02-1/tb02fuchstfm.pdf`.

[3] Donald E. Knuth. *The TeXbook.* Addison Wesley, 1984.

[4] Michael Kohlhase and Dennis Müller. *The sTeX3 Package Collection.* Tech. rep. URL: `https://github.com/slatex/sTeX/blob/main/doc/stex-doc.pdf` (visited on 04/09/2023).

[5] *MathJax: Beautiful Math in all Browsers.* `http://mathjax.org`. URL: `http://mathjax.com`.

[6] Bruce Miller. *LaTeXML: A LaTeX to XML Converter.* URL: `http://dlmf.nist.gov/LaTeXML/` (visited on 03/12/2021).

[7] Dennis Müller. "Mathematical Knowledge Management Across Formal Libraries". PhD thesis. Informatics, FAU Erlangen-Nürnberg, Dec. 2019. URL: `https://opus4.kobv.de/opus4-fau/files/12359/thesis.pdf`.

[8] Dennis Müller and Michael Kohlhase. "sTeX3 – A LaTeX-based Ecosystem for Semantic/Active Mathematical Documents". In: *TUGboat; TUG 2022 Conference Proceedings.* Ed. by Karl Berry. Vol. 43. 2. 2022, pp. 197–201. URL: `https://kwarc.info/people/dmueller/pubs/tug22.pdf`.

[9] Luca Padovani. "MathML Formatting with TeX Rules, TeX Fonts, and TeX Quality". In: *Communications of the TeX Users Group (TUGboat).* Vol. 24. 1. 2003, pp. 53–61. URL: `https://tug.org/tugboat/tb24-1/padovani.pdf`.

[10] *Pandoc – a universal document converter.* `https://pandoc.org/`. 2023.

[11] *sLaTeX/RusTeX.* URL: `https://github.com/sLaTeX/RusTeX` (visited on 04/22/2022).

[12] *TeX4ht.* `https://tug.org/tex4ht/`. URL: `https://tug.org/tex4ht/`.