The crush package

Jesse A. Tov
tov@ccs.neu.edu

This document corresponds to crush v0.2, dated 2011/07/29.

Contents

1 Introduction

The purpose of this package is to provide several methods for making boxes smaller, which extend (and someone overlap with) LaTeX's `\llap` and `\rlap` commands. Most provided commands deal with making boxes of width `0pt`, while anchoring the box in a specified place. For example, consider the following:

To get...	*type...*
()Hello, world!	`(\crushl{Hello, world!})`
Hello, world!()	`(\crushr{Hello, world!})`
Hello,()world!	`(\crushc{Hello, world!})`
$()3x^2 + 4x - 2$	`$(\crushl{3x^2 + 4x - 2})$`
$3x^2 + 4x - 2()$	`$(\crushr{3x^2 + 4x - 2})$`
$3x^2 +()4x - 2$	`$(\crushc{3x^2 + 4x - 2})$`
$3^{x+y}+z$	`$3^{\crushl{x + y}} + z$`
Hello!	`\fbox{\crushl[1em]{Hello!}}`

There is also a command for minimizing the width of a box subject to not increasing its height. For example, to get this:

> Whereas recognition of the inherent dignity and of the equal and inalienable rights of all members of the human family is the foundation of freedom, justice and peace in the world, ...

…write this:

```
\mbox{}\hfill\shrinkbox{\raggedleft
Whereas recognition of the inherent dignity and of the equal and
inalienable rights of all members of the human family is the
foundation of freedom, justice and peace in the world, \ldots}
```

In this case, `\shrinkbox` found the narrowest box in which the given text fits on 3 lines, since given the space available it could not fit on *fewer* than 3 lines.

2 Command Reference

```
\crushl [⟨dimen⟩] {⟨text⟩}
\crushr [⟨dimen⟩] {⟨text⟩}
\crushc [⟨dimen⟩] {⟨text⟩}
```

These commands typeset ⟨*text*⟩ in a horizontal box with width ⟨*dimen*⟩, which defaults to `0pt`. If the natural size of ⟨*text*⟩ exceeds ⟨*dimen*⟩, then the text will extend beyond the box, which means it is likely to overlap the surrounding text. The direction of the overhang is determined by the choice of command:

`\crushl` anchors the *left* edge of the text to the left edge of the box, which may cause it to hang out to the right.

`\crushr` anchors the *right* edge of the text to the right edge of the box, which may cause it to hang out to the left.

`\crushc` anchors the *center* of the text to the center of the box, which may cause the text to hang out to both sides.

```
\uncrushl [⟨dimen⟩] {⟨text⟩}
\uncrushr [⟨dimen⟩] {⟨text⟩}
```

These commands kern by the width of ⟨*text*⟩, adjusted by ⟨*dimen*⟩, which defaults to `0pt`. In particular, `\uncrushl[`⟨*dimen*⟩`]{`⟨*text*⟩`}` moves to the left by the width of ⟨*text*⟩ less `[`⟨*dimen*⟩`]`; `\uncrushr[`⟨*dimen*⟩`]{`⟨*text*⟩`}` moves to the right by the width of ⟨*text*⟩ plus `[`⟨*dimen*⟩`]`. (`\uncrushr{`⟨*text*⟩`}` is equivalent to `\phantom{`⟨*text*⟩`}`.)

```
\vcrush [⟨pos⟩] {⟨width⟩} {⟨text⟩}
```

This command is for crushing vertical-mode text. It sets ⟨*text*⟩ in a box of width ⟨*width*⟩ (in the style of the `minipage` environment). It then crushes the box to width and height `0px`. The ⟨*pos*⟩ argument specifies where with respect to the text the new baseline of the box should be. It accepts all the same positions as `minipage`, and an additional one: `T`, which puts the baseline at the top of the first line of text in the box (whereas `t` uses the baseline of the first line in the box as the baseline of the box).

`\shrinkbox [`⟨*pos*⟩`] [`⟨*dimen*⟩`] {`⟨*text*⟩`}`

This command typesets ⟨*text*⟩ in the narrowest box such that its height does not increase. The optional argument ⟨*dimen*⟩ provides the maximum width for the box, which otherwise defaults to `\linewidth`. This provides a minimal height for the box, and the width is then minimized until making it narrower still would increase the height. This may evaluate ⟨*text*⟩ several times, so any side effects may happen an arbitrary number of times.

The optional argument `[`⟨*pos*⟩`]` gives the vertical position of the text in the box, in the style of `\parbox`.

`\textcrushl [`⟨*dimen*⟩`] {`⟨*text*⟩`}`
`\textcrushr [`⟨*dimen*⟩`] {`⟨*text*⟩`}`
`\textcrushc [`⟨*dimen*⟩`] {`⟨*text*⟩`}`
`\textuncrushl [`⟨*dimen*⟩`] {`⟨*text*⟩`}`
`\textuncrushr [`⟨*dimen*⟩`] {`⟨*text*⟩`}`

The crushing and uncrushing commands normally select text or math mode automatically, but in case they get confused, these are the same commands specialized for text mode.

`\mathcrushl [`⟨*dimen*⟩`] {`⟨*math*⟩`}`
`\mathcrushr [`⟨*dimen*⟩`] {`⟨*math*⟩`}`
`\mathcrushc [`⟨*dimen*⟩`] {`⟨*math*⟩`}`
`\mathuncrushl [`⟨*dimen*⟩`] {`⟨*math*⟩`}`
`\mathuncrushr [`⟨*dimen*⟩`] {`⟨*math*⟩`}`

These are the commands specialized for math mode.

3 Implementation

3.1 Crushing Boxes

`\crusher` A box in which to save stuff to crush:

```
\newsavebox{\crusher}
```

`\crushl` `\crushr` `\crushc` `\uncrushl` `\uncrushr` The main horizontal-mode crushing commands dispatch based on whether we're currently in math mode or text mode:

```
\newcommand\crushl{{%
  \ifmmode\aftergroup\mathcrushl\else\aftergroup\textcrushl\fi
}}
```

```
\newcommand\crushr{{%
  \ifmmode\aftergroup\mathcrushr\else\aftergroup\textcrushr\fi
}}
\newcommand\crushc{{%
  \ifmmode\aftergroup\mathcrushc\else\aftergroup\textcrushc\fi
}}
\newcommand\uncrushl{{%
  \ifmmode\aftergroup\mathuncrushl\else\aftergroup\textuncrushl\fi
}}
\newcommand\uncrushr{{%
  \ifmmode\aftergroup\mathuncrushr\else\aftergroup\textuncrushr\fi
}}
```

`\mathcrush@helper`
`\m@thcrush@helper`

`\mathcrush@helper{`⟨*cmd*⟩`}{`⟨*math*⟩`}` ⟶ ⟨*cmd*⟩`{$`⟨*style*⟩⟨*math*⟩`$}`, where ⟨*style*⟩ is the current math style.

```
\newcommand\mathcrush@helper[1]{\mathpalette{\m@thcrush@helper{#1}}}
\newcommand\m@thcrush@helper[3]{#1{$#2#3$}}
```

`\mathcrushl`
`\mathcrushr`
`\mathcrushc`
`\mathuncrushl`
`\mathuncrushr`

These are the math versions of the crushing and uncrushing macros, which are called by the main versions when in math mode. They use the text versions to do the actual work, using `\mathcrush@helper` to get the contents back in math mode and in the right size.

```
\newcommand\mathcrushl[1][0pt]{\mathcrush@helper{\textcrushl[#1]}}
\newcommand\mathcrushr[1][0pt]{\mathcrush@helper{\textcrushr[#1]}}
\newcommand\mathcrushc[1][0pt]{\mathcrush@helper{\textcrushc[#1]}}
\newcommand\mathuncrushl[1][0pt]{\mathcrush@helper{\textuncrushl[#1]}}
\newcommand\mathuncrushr[1][0pt]{\mathcrush@helper{\textuncrushr[#1]}}
```

`\textcrushl`

This is the implementation of `\crushl` for text mode. It sets the text in a box, drops the box in it right away, then kerns backward by its width and adjusts by any kern requested in the optional argument:

```
\newcommand\textcrushl[2][0pt]{%
  \sbox{\crusher}{#2}%
  \usebox\crusher
  \kern-\wd\crusher
  \kern#1%
}
```

`\textcrushr`

This is the implementation of `\crushr` for text mode. It sets the text in a box, kerns backward by its width, adjusts by any kern requested in the optional argument, and then drops in the box:

```
\newcommand\textcrushr[2][0pt]{%
  \sbox{\crusher}{#2}%
  \kern-\wd\crusher
  \kern#1%
  \usebox\crusher
}
```

\crush@textcrushclen
\textcrushc

For \crushc we need to do half of the adjustment on each side of actually using the box. We use a dimension register to parse any user-specified adjustment so that we can then multiply that by 0.5.

```
\newlength{\crush@textcrushclen}
\newcommand\textcrushc[2][0pt]{%
  \sbox{\crusher}{#2}%
  \setlength{\crush@textcrushclen}{#1}%
  \addtolength{\crush@textcrushclen}{-\wd\crusher}%
  \kern0.5\crush@textcrushclen
  \usebox\crusher
  \kern0.5\crush@textcrushclen
}
```

\textuncrushl
\textuncrushr

For uncrushing, we just measure the text and then kern either its width or the negation of its width:

```
\newcommand\textuncrushl[2][0pt]{%
  \sbox{\crusher}{#2}%
  \kern-\wd\crusher
  \kern#1%
}
\newcommand\textuncrushr[2][0pt]{%
  \sbox{\crusher}{#2}%
  \kern\wd\crusher
  \kern#1%
}
```

\vcrush

This is a little more complicated, as we have to handle the T position ourselves, and its necessary to deal with both width and height.

```
\newcommand\vcrush[3][c]{%
```

Start by setting the given text in a minipage and saving that in a box. We use the position and width specified by the given arguments.

```
  \sbox\crusher{%
    \begin{minipage}[#1]{#2}%
      #3%
    \end{minipage}%
  }%
```

Now we're going to create a second box, setting its width again as specified, but we'll use \vskips to adjust the height:

```
  \sbox\crusher{%
    \vbox{%
      \setlength{\hsize}{\wd\crusher}%
      \ifx T#1\relax
```

For T, we drop in the box and then skip back upward by both its depth and height, which effectively moves the baseline to the top of the box:

```
        \usebox\crusher
        \vskip-\ht\crusher
        \vskip-\dp\crusher
      \else
```

For anything but `T`, `minipage` already put the baseline in the right place, so we adjust away the height of the box before dropping in the box and the depth afterward:

```
        \vskip-\ht\crusher
        \usebox\crusher
        \vskip-\dp\crusher
      \fi
    }%
  }%
  \usebox\crusher
}
```

3.2 Shrinking Boxes

We use binary search on the width of the box, under the constraint that the height does not increase.

`\shrinkboxheighttolerance`
`\shrinkboxwidthtolerance`

First, we define the tolerances for the search. We default to a height tolerance of `0.5ex`, because different line breaking may cause slight adjustments in the height of a box without changing the number of lines in the box. The width tolerance of `1pt` means that we should find a box within `1pt` of the narrowest possible box.

```
\newlength{\shrinkboxheighttolerance}
\newlength{\shrinkboxwidthtolerance}
\setlength{\shrinkboxheighttolerance}{0.5ex}
\setlength{\shrinkboxwidthtolerance}{1pt}
```

`\@shrink@box@a`
`\@shrink@box@b`

We'll use two boxes in our binary search. At any given time, `\@shrink@box@a` will be narrower than `\@shrink@box@b`. We also maintain the invariant that `\ht\@shink@box@b` doesn't increase above the initial height of the maximum width box.

```
\newsavebox{\@shrink@box@a}
\newsavebox{\@shrink@box@b}
```

`\@shrink@box@ht`
`\@shrink@box@wd`

These are temporaries for when we have to measure and compare boxes:

```
\newdimen\@shrink@box@ht
\newdimen\@shrink@box@wd
```

`\shrinkbox`

We need to handle two optional arguments. Here we check for the first, ⟨*pos*⟩, and dispatch to `\shrinkbox@pos` to receive it if it is supplied, or default it to `c` and the width to `\linewidth`, otherwise.

```
\newcommand\shrinkbox{%
  \@ifnextchar [
    \shrinkbox@pos
    {\shrinkbox@start{c}{\linewidth}}%
}
```

`\shrinkbox@pos`

Here we get the optional argument ⟨*pos*⟩ and check if there's a second, which would be ⟨*width*⟩. If the second optional argument isn't supplied, the default is `\linewidth`.

```
\def\shrinkbox@pos[#1]{%
  \@ifnextchar [
    {\shrinkbox@width{#1}}
    {\shrinkbox@start{#1}{\linewidth}}%
}
```

`\shrinkbox@width`

Get the second optional argument.

```
\def\shrinkbox@width#1[#2]{%
  \shrinkbox@start{#1}{#2}%
}
```

`\shrinkbox@start`
`\shrink@box@kont`

Here we initialize the parameters for the binary search. We start the maximum width as the supplied ⟨*width*⟩ (which defaults to `\linewidth`, and try setting the text with that width and $\frac{1}{10}$ of that width. We then start the loop, passing it ⟨*pos*⟩ and ⟨*text*⟩, since we will likely have to set the text again.

```
\newcommand\shrinkbox@start[3]{%
  \setlength{\@shrink@box@wd}{#2}%
  \sbox\@shrink@box@a{\parbox[#1]{0.1\@shrink@box@wd}{#3}}%
  \sbox\@shrink@box@b{\parbox[#1]{\@shrink@box@wd}{#3}}%
  \def\shrink@box@kont{\shrink@box@loop{#1}{#3}}%
  \shrink@box@kont%
}
```

`\shrinkbox@loop`

This is the main loop for the binary search.

```
\newcommand\shrink@box@loop[2]{%
```

Get the differences of heights and widths of the two boxes into the two dimension registers. (We rely on the invariant that assume that box `a` is (non-strictly) narrower and taller than box `b`.)

```
  \setlength{\@shrink@box@ht}{\ht\@shrink@box@a}%
  \addtolength{\@shrink@box@ht}{\dp\@shrink@box@a}%
  \addtolength{\@shrink@box@ht}{-\ht\@shrink@box@b}%
  \addtolength{\@shrink@box@ht}{-\dp\@shrink@box@b}%
  \setlength{\@shrink@box@wd}{\wd\@shrink@box@b}%
  \addtolength{\@shrink@box@wd}{-\wd\@shrink@box@a}%
```

Check if the heights of the two boxes are within the tolerance. If they are, then we should search narrower, but if the heights are very different, this means the narrow box is too narrow.

```
  \ifdim\@shrink@box@ht<\shrinkboxheighttolerance
```

Check the widths are within the tolerance. If they are, then the search is done, since the two boxes have met.

```
    \ifdim\@shrink@box@wd<\shrinkboxwidthtolerance
```

\shrink@box@kont We set \shrink@box@kont to what we want to do next, which is to use the smaller box (though it shouldn't matter, since they're the same size):

```
114       \def\shrink@box@kont{\mbox{\usebox\@shrink@box@a}}%
```

Here the heights are the same but the width are different, so we need to make the wide box narrower. We begin by getting the mean of the width of the boxes in \@shrink@box@wd:

```
115     \else
116       \setlength{\@shrink@box@wd}{0.5\@shrink@box@wd}%
117       \addtolength{\@shrink@box@wd}{\wd\@shrink@box@a}%
```

Then replace the context of the wider box with a new box of the average width:

```
118       \sbox\@shrink@box@b{\parbox[#1]{\@shrink@box@wd}{#2}}%
119     \fi
```

Here the heights are different, so the narrower box needs to get wider. Again we get the mean box width, but we use it to replace the narrower box.

```
120   \else
121     \setlength{\@shrink@box@wd}{0.5\@shrink@box@wd}%
122     \addtolength{\@shrink@box@wd}{\wd\@shrink@box@a}%
123     \sbox\@shrink@box@a{\parbox[#1]{\@shrink@box@wd}{#2}}%
124   \fi
```

Back in \shrinkbox@start, we initialized \shrink@box@kont to run the loop each time. Here, it will recur unless we've determined that it's time to stop and redefined it to actually output the box.

```
125   \shrink@box@kont
126 }
```

Change History

v0.2
General: Initial documented release 1

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.