Ysue

Ashton Wiersdorf

April 20, 2024

Contents

ysue - You Should be Using Emacs

Synopsis

Building:

```
stack build
```

Running the program once `ysue` is on your `PATH`:

```
ysue              -- open up an empty buffer
ysue <FILENAME>   -- edit a file (WARNING: does not work yet!!)
```

Description

Ysue (*yoo*-sway) is a simple text editor implemented in Haskell. Of course, if you intend to do serious text editing, You Should be Using Emacs instead.

Using

```
stack run
```

Ysue has two primary vi-like modes: Insert and Normal. The mode will be displayed at the bottom on the mode line.

Normal Mode

`i` enter Insert Mode

`hjkl` move cursor

`:` enter Extended Command

`q` quit

`Ctrl-e` scroll down

`Ctrl-y` scroll up

Unlike vi/vim, Ysue lets you move the cursor to the end of the line. No need for an "append" command.

Insert Mode

`Esc` return to Normal Mode

any letter key, `backspace`, etc. edit text

Extended Commands

Enter an extended command from Normal Mode by pressing `:`; enter the command in the minibuffer

`e filename.txt` create a new buffer visiting `filename.txt`; can be existing or new file

`w outfile.txt` write buffer contents to `outfile.txt`

`w` write buffer contents to currently visited file

`k` kill current buffer

`n newbuffer` create a new buffer named `newbuffer` (no visiting file associated)

`q` quit

`Ctrl-g` cancel and return to normal mode

Implementation

General

Ysue makes use of the vty package. I am not an expert at terminal handling, and there is a fair amount of jankiness around terminal handling. In particular there are some bugs that crop up with scrolling. Any help welcome.

There is a global editor state `EditorState` which tracks a set of buffers, the terminal height and width, and the current editor mode.

Each buffer is tracked in a `BufferState` struct which carries the current point (cursor position), the point at the top of the screen (for scrolling), buffer contents, and the (Maybe) visited file.

The main loop works like so:

① Transform the `EditorState` struct into a matrix of strings.

② Write these strings onto the terminal.

③ Block until we get an event from the terminal (e.g. a key press)

④ Run the event interpreter, which takes the current `EditorState`, the terminal event, and returns a *new* `EditorState`.

⑤ Goto 1.

This is how we maintain editor state in Haskell. If we wanted to add an undo command, we could theoretically do that by keeping a stack of the last n `EditorState` structs and pop them off at will. I just haven't gotten around to doing something like that. Yay for functional programming!

Text data structure

Ysue uses a rope under the hood to manage arbitrary insertions/deletions into a long string of text.

I ran some benchmarks comparing how a rope vs. a naïve string implementation handle a series of random edits. Results:

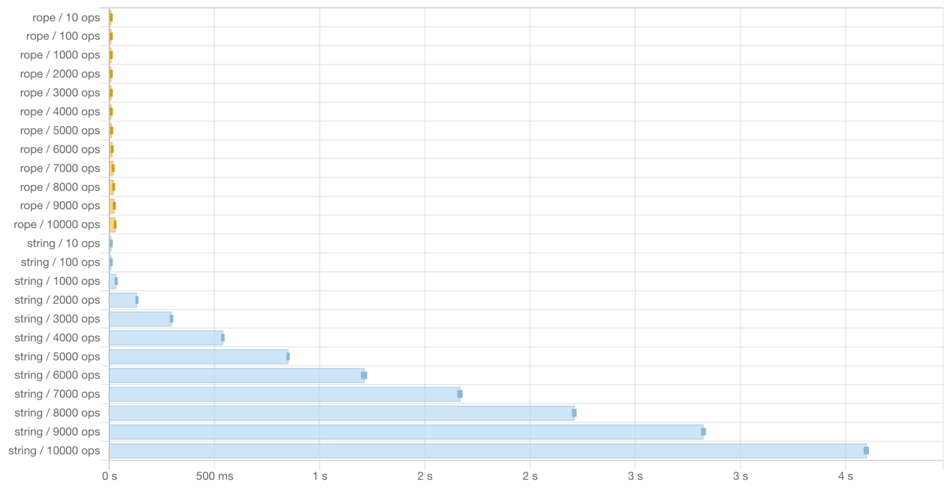

Figure 1: Runtimes of performing n random text editing operations; orange is a rope, blue is a naïve string. Naïve string growth is $O(2^n)$, while rope stays $O(\log(n))$.

Ropes are neat because they leverage structural sharing: i.e. if you edit some text, most of the text that doesn't get edited will stay the same in memory. The garbage collector will clean up the old nodes eventually. However, if you hang onto those nodes (e.g. in an undo stack) then they'll stay around as long as you need them to and they will always point to the same string.

Contributing and Reporting Bugs

This is not a serious project. Really—go use Emacs, Vim, Nano, etc. instead. This is a project for a class and an exercise for me in functional programming in unusual contexts.

If you find a bug, you can report it on the issue tracker on Codeberg. Better yet, if you have a pull request, either open it on Codeberg or fork the project to whatever forge you use, and send me an email at codeberg@wiersdorf.dev.

There are many bugs/janky behaviors around terminal display. If you have any suggestions, I would welcome them.

License

MIT License

Author

Ashton Wiersdorf ◊ https://lambdaland.org/