CS 583 Project Report

Alex Grejuc

June 11th 2020

1 Overview

This project is a modal text editor which runs in a terminal emulator. Its purpose is to bring the keyboard-oriented modal approach of editors such as Vim and kakoune to the domain of literature composition. It is also intended to provide a distraction-free writing experience, similar to FocusWriter. The target users are the rather small group of people who enjoy the power and efficiency of modal text editors but are unwilling to put in the effort to configure them to work well with words, clauses, sentences, paragraphs, or other units of text found in literature composition.

The project provides the crucial features (inserting, removing, or changing text and screen line navigation) and has some proof-of-concept infrastructure in place which specifically targets the domain of literature composition. Some of these more specific features include selecting words, sentences, paragraphs, and changing the case of a selection. Any of the text transformation features exported by `Data.Text` (such as intersperse or reverse) could be easily added by mapping a keypress to application of that function on the current selection.

The application proceeds in the following manner: first, it opens a file specified by a command-line argument in the `Main` module and parses it into a `Zipper`. Then it provides the `Zipper` to the `Editor`, which runs a loop of listening for user input, mapping it to the corresponding `Zipper` function, and rendering the visible portion on the screen. The loop ends when a user chooses to quit, at which point it saves to the command line argument path and then finishes execution.

2 Application Modules

2.1 Zipper

- `Zipper`: This data structure is the text buffer. All user interactions which navigate or edit the text are reflected in the code as modifications to this data structure. It is discussed in detail in Section 3.
- fromText: Makes a `Zipper` from `Text` by splitting on newlines.
- toText3: Converts the visible portion of a Zipper into the text before the selection, the selection, and the text after the selection. This is used for rendering.
- takeTop & takeBottom: Both take n lines of length w from the text which precedes/follows the first/last line of the current selection.
- step functions (e.g. stepLeft): These shift the cursor one column or row. The vertical movements are as the user sees them, not over newlines. Horizontal movements can step over paragraph boundaries and all movements are no-ops if they are at a first or last position.
- downPar & upPar: These shift the cursor vertically through paragraphs. The step functions use them at the row/column extremities for a paragraph.
- position functions (e.g. cursorStart): These provide absolute row and/or column indices which are calculated using screen line lengths. They are used for rendering.

- text editing functions: These are split, backspace, delete, appendChar, appendText, toUpper, and toLower. They operate on the selection. Split inserts a newline just before the start of the cursor and the rest are self-evident.
- selection functions (e.g. selectWord): These all operate in the current paragraph only and modify what is included in the selection.

2.2 Editor

- `MonadEditor`: the monad transformer stack, which implements state and the `Curses` monad. Note that `Cursres` is itself a wrapper around the `IO` monad, with additional ncurses functionality. All functions which directly affect the editor run in this stack.
- `TedState`: stores the state of the editor, which consists of the mode, the offset in screen lines from the beginning of the text, the `Zipper` itself, the path of the file to save to, and a clipboard. Functions running within a `MonadEditor` may access this with the state monad.
- runEditor: renders the buffer and then runs the event loop.
- eventLoop: listens for an event and handles it.
- editEvent: handles an edit mode event by making the corresponding zipper update and displaying it.
- insertEvent: handles an insert mode event by making the corresponding zipper update and displaying it.
- tedRender: renders the visible portion of the buffer and updates the offset if the cursor moves past the top or bottom of the screen.
- draw: is a helper tedRender and does the bulk of the rendering work. It draws the buffer to the screen and highlights the selected text.
- write: overwrites a file at a specified path with the contents of a `Zipper`.

2.3 Main

- start: looks for exactly one command line argument that is the path of the file which is to be edited and returns the path and contents of the file. Returns empty `Text` if the file does not exist and exits the program if exactly one argument was not supplied.
- main: uses start to set up the editor with an initial state and then provides it to runEditor

3 Design Decisions

3.1 The Zipper

The most significant design decision in this project is the choice of data structure for the text buffer. In an imperative setting with ephemeral data structures, a constant-time random access container of strings might have been a suitable and simple first approach to store lines of text. Considering that text editing mostly involves small local updates to a single buffer, such an editor could be practically useful.

However, due to persistence, this would not be a suitable approach in a pure functional language such as Haskell. Changes to the buffer require copying the nodes along the path to the update location. A particularly bad approach would have been `[String]`. In this case, updates would be linear in time and space on the number of lines. Even a container with faster access, such as `Seq String` still suffers the penalty of copying all the lines of text needed to access a particular line for each update.

To remedy this, I used the zipper technique. A zipper is a way to structure data with a notion of locality. That is, it makes use of a focus which can be accessed and updated without traversing the rest of the data structure. It also maintains a context which can be combined with the focus to reconstruct that which it represents. Additionally, the time to access data is a function of its distance from the focus. Thus, if the common operations on the data structure occur locally, the traversals are small and thus minimal copying occurs.

In concrete terms, I used a nested zipper of `Text`. I based this off of the suggestions provided by Peter Pudlák. The Data definition is shown below.

```
-- | A 2-d zipper representing the contents of a text file with newlines stripped.
--   A paragraph is represented by the Text to the left of the currently-selected
--   contents, and the Text to its right.
data Zipper =
   Zipper { above     :: Seq Text   -- ^ The paragraphs of text above the current paragraph
          , left      :: Text       -- ^ The text left and above the current selection
          , selection :: Text       -- ^ The current selection
          , right     :: Text       -- ^ The text right and below the current selection
          , below     :: Seq Text   -- ^ The paragraphs of text below the current paragraph
          }
   deriving (Show, Eq)
```

Listing 1: The Zipper data type, which buffers the text being edited

Each `Text` above and below represents a paragraph of text and the current paragraph made up of *left, selection,* and *right* is the focus. With this setup, updates are made to the current paragraph without traversing any other paragraphs, thus the rest of the buffer is not copied. It is as if we were making updates to an ephemeral data structure!

The current paragraph is further structured as a sub-zipper in which the selection is the focus and the surrounding text is the context. In practice, it likely would have been feasible to have a single `Text`. However, the sub-zipper is more efficient for changing text and it also obviates the need for extra state to track the position of the selection. Finally, it simplifies the code to extract and modify the selection. There are some trade offs to this substructure in comparison to a single `Text`, however. Moving the selection within a paragraph could have been reduced to updating indices, and moving up or down a paragraph would not require concatenating three `Texts`.

3.2 State Minimization & Composability

I intentionally made the `Zipper` as stateless and minimal as possible. Information about it is computed on the fly, which means that it does not need to be changed if the representation is changed (such as a screen resizing). I also intentionally separated it into its own module which does not depend on the editor. By keeping it completely separate, a large portion of the code is pure. One result of this is that it is easy to extend the `Zipper` transformation functions. When writing these functions, one only needs to worry about rearranging or editing text without any concern for maintaining a correct state. This is generally in line with the philosophy of functional programming. The functions on the zipper are short, simple, and pure. This approach also led to a small combinator library. These could be used to build an editing language in with a `Zipper -> Zipper` semantic domain since the functions are easily composable operations (although currently their composition is not especially useful).

3.3 Monad Transformer

I chose to use monad transformers to combine IO (via the `Curses` monad) and state. IO is a necessity for the project, but I had the alternative of threading the state through all the functions which need it. The benefit of this is not massive for the project, but it does clean up function signatures and reduce the number of parameters for many of them. For example, the current signature for editEvent is

```
editEvent :: Window -> Event -> MonadEditor ()
```

and it would have had to be:

```
editEvent :: Window -> Event -> TedState -> Curses ()
```

One more significant advantage of using monad transformers is that the editor can more easily be extended with additional monads if they were needed.

3.4 Typeclass

A choice I knowingly did not make was to make a more generic `Zipper` or even a `Zipper` type class. The advantage of this would have been the normal advantages that abstraction provides: text zippers on different containers and string-like data types could more easily be created and plugged into the editor. The reason I did not do this is because it would have made it more difficult to write the code. I was engaging with enough new concepts to be satisfied with a concrete data structure.