CSCI 365 Problem Set 7: Type Classes & Monoids

due Wednesday, 13 March 2024

Specification

To receive credit for this problem set:

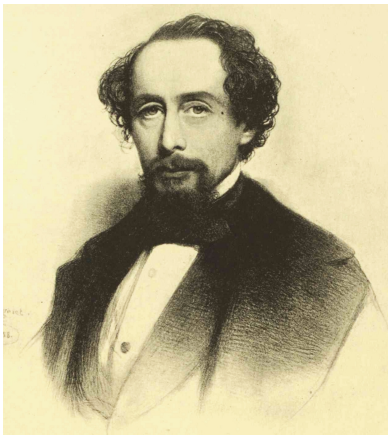
- Complete at least 10 out of 13 exercises.
- Submit a `.zip` or `.tgz` file containing all the files necessary for running the code (including the all the files provided in the skeleton package as well as any additional files you created).
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Haskell code should be submitted in one or more `.hs` files. Written exercises may be submitted either as a PDF, or as comments in one of the `.hs` files.

The First Word Processor

As everyone knows, Charles Dickens was paid by the word.[1] What most people don't know, however, is the story of you, the trusty programming assistant to the great author.

[1] Actually, this is a myth.



In your capacity as Dickens's assistant, you program and operate the steam-powered Word Processing Engine which was given to him as a thoughtful birthday gift from his friend Charles Babbage.[2] To be helpful, you are developing a primitive word processor the author can use not only to facilitate his craft, but also to ease the accounting.[3] What you have done is build a word processor that keeps

[2] Unlike the rest of this story, the fact that Dickens and Babbage were friends is 100% true. If you don't believe it, just do a Google search for "Dickens Babbage".

[3] Of course, all of your programming is actually done by assembling steam pipes and valves and shafts and gears into machines which perform the desired computations; but as a mental shortcut you have taken to *thinking* in terms of a higher-order lazy functional programming language and compiling down to steam and gears as necessary. In this assignment we will stick to the purely mental world, but of course you should keep in mind that we could compile everything into Steam-Powered Engines if we wanted to.

track of the total number of words in a document while it is being edited. This is really a great help to Mr. Dickens as the publishers use this score to determine payment, but you are not satisfied with the performance of the system and have decided to improve it.

Getting Started

Download the provided skeleton code from `https://hendrix-cs.github.io/csci365/problem-sets/07-monoids/dickens-editor.tgz` and unpack it somewhere. Inside the package you should find the following:

- `dickens-editor.cabal` is a Cabal package file describing this Haskell package. Edit the `author`, `maintainer`, and `copyright` fields with your name and email.
- `LICENSE` describes the license under which this package can be distrubuted. At the very least, add your name to the copyright notice in this file. The BSD 3-clause license is common in the Haskell open-source community, but you can also use a different license if you wish.
- `carol.txt` contains the entire text of "A Christmas Carol" by Charles Dickens. You are welcome to add other texts to try out as well.
- There should also be some `.hs` files in the `lib` subdirectory, and one in `app`, which will be explained throughout the rest of the problem set.

Check out Project Gutenberg at `https://www.gutenberg.org/` for a great source of works in the public domain.

Editors and Buffers

You have a working user interface for the word processor implemented in the file `lib/Editor.hs`. The `Editor` module defines functionality for working with documents implementing the `Buffer` type class found in `lib/Buffer.hs`. Take a look at `Buffer.hs` to see the operations that a document representation must support to work with the `Editor` module. The intention of this design is to separate the front-end interface from the back-end representation, with the type class intermediating the two. This allows for the easy swapping of different document representation types without having to change the `Editor` module.

Try out the editor by typing

```
cabal run stringbufeditor
```

at a command line (make sure you are in the directory with the `.cabal` file first). The editor interface is as follows:

- `v` — view the current location in the document
- `n` — move to the next line
- `p` — move to the previous line
- `l` — load a file into the editor
- `e` — edit the current line
- `q` — quit
- `?` — show this list of commands

To move to a specific line, enter the line number you wish to navigate to at the prompt. The display shows you up to two preceding and two following lines in the document surrounding the current line, which is indicated by an asterisk. The prompt itself indicates the current value of the entire document.

The first attempt at a word processor back-end was to use a single `String` to represent the entire document. You can see the `Buffer` instance for `String` in the file `lib/StringBuffer.hs`. Performance isn't great because reporting the document score requires traversing every single character in the document every time the score is shown! Mr. Dickens demonstrates the performance issues with the following (imaginary) editor session:

```
$ runhaskell StringBufEditor.hs
33> n
 0: This buffer is for notes you don't want to save, and for
*1: evaluation of steam valve coefficients.
 2: To load a different file, type the character L followed
 3: by the name of the file.
33> l carol.txt
31559> 3640
 3638:
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
*3640: Do you know whether they've sold the prize Turkey that
 3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
31559> e
Replace line 3640: Do you know whether they've sold the prize Goose that
31559> n
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
```

```
*3641: was hanging up there?--Not the little prize Turkey: the
 3642: big one?"
 3643:
31559> e
Replace line 3641: was hanging up there?--Not the little one: the
31558> v
 3639: "An intelligent boy!" said Scrooge. "A remarkable boy!
 3640: Do you know whether they've sold the prize Goose that
*3641: was hanging up there?--Not the little one: the
 3642: big one?"
 3643:
31559> q
```

Sure enough, there is a small delay every time the prompt is shown.

You have chosen to address the issue by implementing a light-weight, tree-like structure, both for holding the data and caching the metadata. This data structure is referred to as a *join-list*. A data type definition for such a data structure might look like this:

```
data JoinListBasic a where
  Empty   :: JoinListBasic a
  Single  :: a -> JoinListBasic a
  Append  :: JoinListBasic a -> JoinListBasic a -> JoinListBasic a
```

The intent of this data structure is to directly represent append operations as data constructors. This has the advantage of making append an `O(1)` operation: sticking two `JoinLists` together simply involves applying the `Append` data constructor.

Such a structure makes sense for text editing applications as it provides a way of breaking the document data into pieces that can be processed individually, rather than having to always traverse the entire document. This structure is also what you will be annotating with the metadata you want to track.

Monoidally Annotated Join-Lists

The `JoinList` definition that we will actually use for this assignment is

```
data JoinList m a where
  Empty   :: JoinList m a
  Single  :: m -> a -> JoinList m a
  Append  :: m -> JoinList m a -> JoinList m a -> JoinList m a
  deriving (Eq, Show)
```

This definition has been provided for you in `lib/JoinList.hs`.

The `m` parameter will be used to track monoidal annotations to the structure. The idea is that the annotation at the root of a `JoinList` will always be equal to the combination of all the annotations on the `Single` nodes (according to whatever notion of "combining" is defined for the monoid in question). In other words:

- `Empty` nodes do not explicitly store an annotation, but we consider them to have an annotation of `mempty` (that is, the identity element for the given monoid).
- `Single` nodes store some `m` value corresponding to the stored data of type `a`.
- `Append` nodes just store the combination of the `m` values in their children.

For example,

```
Append (Product 210)
  (Append (Product 30)
    (Single (Product 5) 'y')
    (Append (Product 6)
      (Single (Product 2) 'e')
      (Single (Product 3) 'a')))
  (Single (Product 7) 'h')
```

is a join-list storing four values: the character `'y'` with annotation `5`, the character `'e'` with annotation `2`, `'a'` with annotation `3`, and `'h'` with annotation `7`. (See Figure 1 for a graphical representation of the same structure.) Since the multiplicative monoid is being used, each `Append` node stores the product of all the annotations below it. The point of doing this is that all the subcomputations needed to compute the product of all the annotations in the join-list are cached. If we now change one of the annotations, say, the annotation on `'y'`, we need only recompute the annotations on nodes above it in the tree. In particular, in this example we don't need to descend into the subtree containing `'e'` and `'a'`, since we have cached the fact that their product is `6`. This means that for balanced join-lists, it takes only $O(\log n)$ time to rebuild the annotations after making an edit.

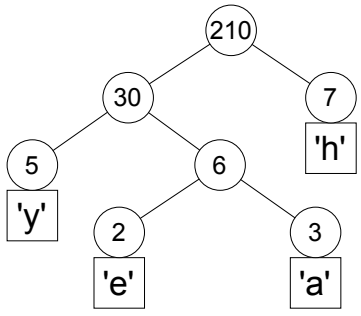


Figure 1: A sample join-list annotated with products

Exercise 1 We first consider how to write some simple operations on these `JoinLists`. Perhaps the most important operation we will consider is how to append two `JoinLists`. Previously, we said that the point of `JoinLists` is to represent append operations as data, but what about the annotations? Edit `lib/JoinList.hs` and write an append function for `JoinLists` that yields a new `JoinList` whose monoidal annotation is derived from those of the two arguments.

"`infixr 5 +++`" declares `(+++)` to be right associative with precedence level 5, the same as `(++)`.

```
infixr 5 +++
(+++) :: Monoid m => JoinList m a -> JoinList m a -> JoinList m a
```

You may find it helpful to implement a helper function

```
tag :: Monoid m => JoinList m a -> m
```

which gets the annotation at the root of a `JoinList`.

Exercise 2 We will need to be able to convert a `JoinList` to a regular list. Write a function

```
jlToList :: JoinList m a -> [a]
```

which ignores the `m` annotations and concatenates all the `a` values at the leaves of a `JoinList` into a single list.

Note: the straightforward way to do this takes $O(n)$ on a balanced `JoinList` but can take up to $O(n^2)$ if the `JoinList` is imbalanced. If you wish, you can use the technique from Exercise 7 on Problem Set 4 to write a version which runs in worst-case linear time.

Exercise 3 We will also want to be able to convert in the other direction, from a list into a `JoinList`. Write a function

```
jlFromList :: Monoid m => (a -> m) -> [a] -> JoinList m a
```

which takes as arguments a list and a function for extracting a suitable `m` value for each `a`, and builds a *balanced* `JoinList` containing all the elements of the list in order.

Hint: use divide and conquer. By "balanced" we just mean that each `Append` node should have approximately the same number of elements on both sides.

Exercise 4 The first annotation to try out is one for fast indexing into a `JoinList`. The idea is to cache the *size* (number of data elements) of each subtree. This can then be used at each step to determine if the desired index is in the left or the right branch.

I have provided `lib/Sized.hs` that defines the `Size` type, which is simply a `newtype` wrapper around an `Int`.

Add `import Sized` to the top of `lib/JoinList.hs` (just after the `module` declaration) and then implement a function

```
indexJ :: Int -> JoinList Size a -> Maybe a
```

`indexJ` finds the `JoinList` element at the specified index. If the index is out of bounds, the function returns `Nothing`. By an *index* in a `JoinList` we mean the index in the list that it represents. That is, consider a safe list indexing function

```
(!!?) :: [a] -> Int -> Maybe a
[]     !!? _         = Nothing
_      !!? i | i < 0 = Nothing
(x:xs) !!? 0         = Just x
(x:xs) !!? i         = xs !!? (i-1)
```

which returns `Just` the *i*th element in a list (starting at zero) if such an element exists, or `Nothing` otherwise. Then for any index `i` and join-list `jl`, it should be the case that

```
(indexJ i jl) == (jlToList jl !!? i)
```

That is, calling `indexJ` on a join-list is the same as first converting the join-list to a list and then indexing into the list. The point, of course, is that `indexJ` can be more efficient ($O(\log n)$ versus $O(n)$, assuming a balanced join-list), because it gets to use the size annotations to throw away whole parts of the tree at once, whereas the list indexing operation has to walk over every element.

Hint: when considering an `Append` node, look at the `Size` annotations on its two children. By comparing the desired index to the sizes, you should be able to figure out whether to descend into the left or right subtree. Also note that you can use `getSize` to convert a `Size` into an `Int`.

Exercise 5 Implement functions

```
dropJ :: Int -> JoinList Size a -> JoinList Size a
takeJ :: Int -> JoinList Size a -> JoinList Size a
```

which are analogous to the standard `drop` and `take` functions on lists. Formally, `dropJ` and `takeJ` should behave in such a way that

```
jlToList . dropJ n == drop n . jlToList
jlToList . takeJ n == take n . jlToList
```

Ensure that your function definitions use the `Size` annotations to make smart decisions about how to descend into the `JoinList` tree, similarly to `indexJ`.

Hint: you can either define `dropJ` and `takeJ` individually, or you can try implementing a single function

```
splitAtJ :: Int -> JoinList Size a -> (JoinList Size a, JoinList Size a)
```

and then defining `takeJ` and `dropJ` in terms of `splitAtJ`.

Exercise 6 We are going to want to be able to annotate `JoinLists` with more than just a `Size`; to this end, `lib/Sized.hs` also provides a `Sized` type class which provides a method for obtaining a `Size` from some value.

Generalize `indexJ`, `dropJ`, and `takeJ` so that they work with `JoinLists` with any `Sized` annotation. Instead of directly expecting a `Size` value as an annotation, they can call the `size` function on annotations to get a `Size`. In fact, if you make a `Sized` instance for `JoinList m a` (as long as there is a `Sized m` instance) you can directly call `size` on any `JoinList` (for example, the two children at an `Append` node).

The types should now be:

```
indexJ :: (Monoid m, Sized m) => Int -> JoinList m a -> Maybe a
dropJ :: (Monoid m, Sized m) => Int -> JoinList m a -> JoinList m a
takeJ :: (Monoid m, Sized m) => Int -> JoinList m a -> JoinList m a
```

Since `Size` itself is an instance of `Sized`, these should continue to work with `JoinList Size a`. However, the point is that they will now also work with `JoinLists` with multiple annotations, like `JoinList (Score, Size) a`, which we turn to next.

Exercise 7 Now you need to implement an annotation to keep track of word count. Create a new module in `lib/WordCount.hs` and add it to the `exposed-modules` list in the `.cabal` file. `WordCount.hs` should define a `WordCount` type, `Semigroup` and `Monoid` instances for `WordCount`, and a function `wordCount :: String -> WordCount` to count the number of words in a given `String`.

To test that you have everything working, add the line `import WordCount` to the import section of your `JoinList` module, and write the following function to create a singleton `JoinList` annotated with a word count:

```
countedSingle :: String -> JoinList WordCount String
```

Now you should be able to type `cabal repl` at the command line to load up the project in GHCi, and do things like the following:

```
ghci> :m +JoinList
ghci> countedSingle "hello there" +++ countedSingle "haskell!"
Append (WordCount 3)
       (Single (WordCount 2) "hello there")
       (Single (WordCount 1) "haskell!")
```

Exercise 8 Finally, combine these two kinds of annotations. As we saw in class, a pair of monoids is itself a monoid:

```
instance (Semigroup a, Semigroup b) => Semigroup (a,b) where
  (a1,b1) <> (a2,b2) = (a1 <> a2, b1 <> b2)

instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
```

This means that join-lists can track more than one type of annotation at once, in parallel, simply by using a pair type.

Create a new module `lib/JoinListBuffer.hs` and add it to the list of `exposed-modules` in the `.cabal` file. Using all the functions you have developed so far in the previous exercises, implement a `Buffer` instance for `JoinList (WordCount, Size) String`.

Note that you will probably have to enable the `FlexibleInstances` extension.

Due to the use of the `Sized` type class, this type should continue to work with your generalized `indexJ`, `takeJ`, and `dropJ` functions.

Exercise 9 Finally, we can put everything together to make an editor executable based on `JoinList`, which should behave identically to the original `StringBuffer`-based editor, but much faster.

- Create a new file `app/JoinListEditor.hs` which is identical to `app/StringBufEditor.hs`, except that
  - it should import `JoinListBuffer` instead of `StringBuffer`, and
  - you will have to construct some initial buffer contents of type `JoinList (WordCount, Size) String` instead of `String`. (*Hint*: use `jlFromList` instead of `unlines`. You may want to write a helper function of type `String -> (WordCount, Size)` which returns a the word count of a `String` paired with `Size 1`.)
- Create a new `executable` section in the `.cabal` file which is identical to the existing `executable` section, except that it is called `joinlisteditor` instead of `stringbufeditor`, and uses `JoinListEditor.hs` for `main-is` instead of `StringBufEditor.hs`.

  You should now be able to test out your editor by executing

```
cabal run joinlisteditor
```

at the command line. Verify that the editor demonstration described in the section "Editors and Buffers" works identically but does not exhibit delays when showing the prompt.

Further Exploration

Exercise 10 Mr. Dickens's publishing company has changed their minds. Instead of paying him by the word, they have decided to pay him according to the scoring metric used by the immensely popular game of *Scrabble*™. You must therefore update your editor implementation to count Scrabble scores instead of counting words.

Scrabble™, of course, was invented in 1842, by Dr. Wilson P. Scrabble™.

Create a new module in `lib/Scrabble.hs` and add it to the `exposed-modules` list in the `.cabal` file. `Scrabble.hs` should define a `Score` type, `Semigroup` and `Monoid` instances for `Score`, and the following functions:

```
score :: Char -> Score
scoreString :: String -> Score
```

Hint for `scoreString`: take a look at the standard `foldMap` function.

The `score` function should implement the tile scoring values as shown at `http://www.thepixiepit.co.uk/scrabble/rules.html`; any characters not mentioned (punctuation, spaces, *etc.*) should be given zero points. `scoreString` should simply add up the scores for every character in an entire `String`.

Exercise 11 Instead of implementing explicit `Semigroup` and `Monoid` instances for `WordCount` and/or `Score`, it is possible to get GHC to write this code for us using the `DerivingVia` extension, like so:

```
{-# LANGUAGE DerivingStrategies #-}
{-# LANGUAGE DerivingVia #-}

newtype WordCount = ...
  deriving (Eq, Ord, Show)
  deriving (Semigroup, Monoid) via Sum Int
```

Read about this feature at `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/deriving_via.html` and explain how this works.

Exercise 12 Extend the editor with another feature such as text search or find and replace. Extend the `Buffer` class with appropriate method(s), and implement them for both `StringBuffer` and `JoinListBuffer`, and extend the implementation in `lib/Editor.hs` with a new keyboard shortcut that calls the appropriate `Buffer` methods.

Exercise 13 Read the article "Monoids and Finger Trees" at `https://apfelmus.nfshost.com/articles/monoid-fingertree.html`. Explain how we could use the ideas described there to extend the editor to always keep track of the location of the lexicographically smallest word in the entire document, and update in only $O(\lg n)$ time every time the smallest word is edited or removed.