Functional Programming II

Lukas Pietzschmann

Institute of Software Engineering and Programming Languages

Sheet 4: Lenses

May 17th, 2024

Some useful pointers

If you ever get stuck, you can try helping yourself first. I'll leave you some notes on the documentation here.

You can find the lens package's documentation here. If you scroll down to Modules, you can find an overview of all the modules that are part of the package. Here's a little summary of the most important ones:

Control.Lens.Lens Here you can find the `Lens` type itself and some functions to work with it. We won't need most of them, but you can, e.g., find the &-operators and the `lens` function here.

Control.Lens.Operators Here are all the operators we used. Since this module is only a listing of operators from other modules, you can find hyperlinks to the actual modules on the left or above every section. You can press `<CTRL-f>` on this page, hit random symbols on your keyboard, and you'll probably find an operator with this name.

Control.Lens.Getter You probably alredy expected `to` and `like` to be defined here, but you'll also find the `view` function and its corresponding operator there.

Control.Lens.Setter Analogous to the `Getter` module, you can find `set` and `setting` here. Along with, e.g., `over` and some `set`-like operators.

Control.Lens.Prism Here, all prism related types and functions are defined. You might be primarily interested in the section Common Prisms.

Control.Lens.Traversal Basically, we only used two functions from here: `traverse`, and `over`. But the Common Traversals section might contain some more interesting functions.

If you are looking for a specific function, you can also always use the Quick Jump button on top of the page, or just press `s`. There, you can just type the name of the function and the search box will present you a link to its documentation.

Exercise 0 Setup

To get started, you should have the lens package installed. You can use cabal for this:

```
cabal install --lib lens
```

On my system, I have GHC `9.4.8` and lens `5.2.3` installed, but older versions should also work just fine (famous last words).

You can check if your installation works by loading the provided `tasks.hs` file into GHCi and evaluating `test`.

```
ghci tasks.hs
ghci> test
((1,2),3,4)
```

If this prints the correct result, you should be good to go.

Exercise 1 Getting familiar with lenses

Let's start easy by composing some lenses and accessing and changing values.

a) Combine multiple different lenses to form a new one that focuses on the string `"Hi"` inside `(1, ("Hi", "Ho"), 2)`.

b) Now, use this lens to retrieve the focused part. Try both the operator and function for this.

c) Change `"Hi"` to a integer value of your choice and bind the result to a name. Again, try the operator and function.

d) Finally, use the lens on the updated tuple, to multiply the integer value you set in the last task with `11`.

e) Now we change things up a little bit. The tuple now contains lists of stings, with `"Hi"` being in the first one: `(["Hi", "Ho"], ["He", "Hu"])`. Change your lens so it still focuses on `"Hi"`.

f) Can you think of another lens that can focus on `"Hi"`?

g) Finally, we no longer want to focus on just one element, but on `"Hi"` and `"He"` at the same time. Write a lens that does exactly this.

h) Now use this lens to change both `"Hi"` and `"He"` to a string of your choice.

Exercise 2 We need to go deeper

Now that you feel comfortable with lenses, let's go a little further. In this exercise, we will be working with a file system. I have prepared some ADTs in `tasks.hs`, go take a look at them ... Alright, let's get started.

a) We will start simple by creating a lens for each element of the `Metadata` ADT.

In the following, we will use this as an example:

```haskell
example :: FileSystem
example = Folder "root" [
    File $ Doc Text (Metadata ".zshenv" "root") "",
    Folder "home" [
        Folder "luke" [
            File $ Doc Text (Metadata ".zshenv" "luke") "export EDITOR=nvim",
            File $ Doc Text (Metadata ".zsh_history" "luke") "sudo dnf rm java"
          ]
      ]
  ]
```

b) Without evaluating it, determine what the expression `example ^. _File . metadata` would evaluate to?

c) If you have not figured out the previous task: it does not evaluate to anything, as there's a type error. Implement a fix for it, without changing the expression.

d) How could we have changed the above expression to fix the error, while not changing what the lens focuses on?

Aren't lenses, prisms, and traversals nice? But for the remaining tasks of this exercise, we need a new thing ... `Fold`s! But don't worry, I will gently introduce you to them.

Imagine, you want to modify the contents of a specific file, but you don't know its exact path. The only thing you know is its name. After the following tasks, you will have written functions that can focus on a specific file, no matter its index in the list.

e) To get into the right mood, forget lenses for a moment. Let's start by writing a plain old Haskell function `searchFiles :: String → [Document] → [Document]` using **`filter`**. For now, we assume that any input to this function is a list with one layer of documents:

```haskell
searchFiles ".zshenv" [
  Doc Text (Metadata "lost+found" "root") "",
  Doc Text (Metadata ".zshenv" "luke") "export EDITOR=nvim",
  Doc Text (Metadata ".zsh_history" "luke") "sudo dnf rm java"
]
▸ [Doc Text (Metadata ".zshenv" "luke") "export EDITOR=nvim"]
```

Now, we need a way to handle arbitrary nesting depths. We can tackle this by flattening everything into a single list of files.

Optional *f)* Implement `flattenFolders :: File Document → [Document]`. This function takes a directory tree and flattens it to a list of documents.

Optional *g)* Now, implement `searchFiles' :: String → File Document → [Document]`. This function should work exacly like `searchFiles`, but for directory trees.

Now it's lens time! The cool thing with lenses is, that we don't need to do the flattening explicitly. You already know that `^..` gives us a flat list of all targets of a traversable. So providing a `Traversable` instance for `File` should do the trick. But a `Traversal` also needs to be a `Functor` and `Foldable`. Implementing all those instances looks like a lot of work. Luckily `^..` also works on `Fold`s, and in order to create a `Fold` from `File`, we only need a `Foldable` instance!

h) Implement an instance of `Foldable` for `File` that folds over all documents in the file hierarchy, no matter the depth.

Awesome! Now we can use the `folded` function to build a `Fold` from our `Foldable` and use `^..` to view a flat list of all targets of the fold!

```
example ^.. folded
```

As a last step, we now need a way to filter this list in the lens-way.

i) Look through Control.Lens.Fold and search for a function that can be composed with a `Fold` and be used for filtering.

j) Now use this function like the `filter` from task *e)*, so that we can implement `searchFiles''` and make it work like this:

```
example ^.. folded . searchFiles'' ".zshenv" . content
▸ ["", "export EDITOR=nvim"]
```

By the way, if we are sure that there will be at maximum a single file with this name, we can also use `^?` to get a `Maybe` instead of a list.

```
example ^? folded . searchFiles'' ".zsh_history" . metadata . author
▸ Just "luke"
```

k) With this new superpower unlocked, implement `searchAuthor` in the same way to focus on all files from a specific author.

```
example ^.. folded . searchAuthor "luke" . metadata . title
▸ [".zshenv",".zsh_history"]
```

l) Last but not least, implement `filesWithAuthor` to search for files with a specific author and name.

```
example ^?! folded . filesWithAuthor ".zshenv" "luke" . content
‣ "export EDITOR=nvim"
```

Exercise 3 Reinventing the wheel

In this exercise we want to reimplement some common library function to get a better understanding of how they work.

You might remember our exploration of the `Lens` type from the lecture. Here's a little refresher:

```
type Lens s t a b = forall f. Functor f ⇒ (a → f b) → s → f t
```

Depending on the operation performed on the lenses target, Haskell can select a fitting functor automagically. The functor then carries out the operation.

a) The set-like operator `.~` makes use of the `Identity` functor, so it requires a lens of type `(a → Identity b) → s → Identity t`. Knowing this, reimplement this operator. We'll name it `.~.`[1] and it should have the following type signature:

```
(.~.) :: ((a → Identity b) → s → Indentity t) → b → s → t
```

If you're a bit rusty on how `Identity` works and how you get things out of it, check out its documentation.

b) Next, reimplement the operator for `over`. It also uses the `Identity` functor. Your implementation should have the following type signature:

```
(%~.) :: ((a → Identity b) → s → Identity t) → (a → b) → s → t
```

c) The last set-like operator you want to reimplement is `*~`. It multiplies the lense's target by a given value, and — you already guessed it — also used the `Identity` functor. Here's its signature:

```
(*~.) :: Num a ⇒ ((a → Identity a) → s → Identity t) → a → s → t
```

Set-like function are only one side of the coin. In the next task, we want to tackle `view`, or more precisely, the `^.` operator. Insead of the `Identity` functor, it uses `Const`. Again, here's a link to its documentation, so you can familiarize yourself with it. Operators using `Const`, take lenses looking like this: `(a → Const a b) → s → Const a t`.

[1] It kinda looks like a person with a mustache, but upside down 🥸

d) Implement the view operator `^.` respecting the following type signature:

```
(.^.) :: s → ((a → Const a b) → s → Const a t) → a
```

The other view-like functions and operators are a bit more complex and work trhough way more abstractions, so we'll stop here. Hopefully, this took away the type magic of lenses a bit. If you still have questions, I'll be happy to have a little chat about it.