The Barcode Reader

Informatics 1 – Introduction to Computation
Functional Programming Resit Exam 2022

Orestis Melkonian, Philip Wadler

due 16.00 Friday 12 August 2022

See instructions on Learn for submission details.

This exercise must be completed without consultation with other people.

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

1 The barcode reader

In this exam we will take a look at a barcode scanner. Of course, we will not be doing the actual scanning, but what we *will* do is search a database for the item that belongs to a scanned barcode. We will read the database from a file and store it in different shapes, to see which gives the fastest retrieval times.

The Haskell files that come with this tutorial are `Resit.hs`, `KeymapList.hs`, and `KeymapTable.hs`. There is also the database itself: `database.csv` (`csv` stands for 'comma-separated values').

Let's start by opening `KeymapList.hs`. This file defines an abstract data type for a keymap. The file starts as follows:

```
module KeymapList ( Keymap,
                    size,
                    ...
                    )
where
```

This declaration means that `KeymapList` is a *module* that can be used by other Haskell files, just like `Data.Char` and `Test.QuickCheck`. The functions and constructors mentioned in parentheses (`Keymap`, `size`, etc.) are the ones that are *exported* by the module, i.e. the ones that can be used when the module is imported somewhere else.

Next, let's take a look at the type declaration for `Keymap`.

```
newtype Keymap k a = K [(k,a)]
```


This defines the polymorphic data type `Keymap`. The first argument (`k`), is what's used as the *key* in the keymap, the second (`a`) is the type of the *values* stored. For instance, a keymap of type `Keymap Int String` associates keys of type `Int` with values of type `String`.

Finally, there is the definition itself, `K [(k,a)]`. As you see, a keymap is simply stored as a list of key-value pairs. The type-constructor `K` is just a wrapper that prevents us from using normal list functions on keymaps. This is precisely the idea: we want the type to be *abstract*, so we can hide the fact that we are using lists underneath. This ensures that we don't use ordinary list functions on keymaps and accidentally invalidate the representation. And it frees us to change our representation whilst guaranteeing that our users' code still works.

Now, let's look at the functions in the file. The *constraints* `Eq k` mean that whatever the type `k`, it must support *equality testing*. In other words, if we want to use a type as a key, we have to be able to use the function `(==)` on values of that type—otherwise, we would have no way to identify the right key in the keymap.

- `size :: Eq k => Keymap k a -> Int`

  This function gives the number of entries in a `Keymap`.

- `get :: Eq k => k -> Keymap k a -> Maybe a`

  Given a key, this function looks up the associated value in a `Keymap`. In this function keys are matched using `(==)`, which is why the constraint `Eq k` is needed. The value returned is not just of type `a`, but `Maybe a`, because a key might not occur in a `Keymap`. We will get back to this later.

- `set :: Eq k => k -> a -> Keymap k a -> Keymap k a`

  Given a key and a value, this function sets the value associated with the key to the given value in a `Keymap`. If the key already had a value, this value is replaced; otherwise the key is newly added to the keymap.

- `del :: Eq k => k -> Keymap k a -> Keymap k a`

  This function deletes an entry from a keymap.

- `select :: Eq k => (a -> Bool) -> Keymap k a -> Keymap k a`

  This function narrows a keymap down to those values that satisfy a given predicate.

- `toList :: Eq k => Keymap k a -> [(k,a)]`

  This function exports the keymap as a list.

- `fromList :: Eq k => [(k,a)] -> Keymap k a`

  This function builds a keymap from a list.

Now that we know what `KeymapList` is like, we can start working on `Resit.hs`. Just below the top, you will find the declarations:

```
import KeymapList

type Barcode = String
type Name    = String
type Price   = Int -- in pence

type Item = (Name, Price)

type Catalogue = Keymap Barcode Item
```


Firstly, we are importing the `KeymapList` module. Next, we define a `Catalogue` to be a map associating a `Barcode` to an `Item`, each having a `Name` and a `Price` **in pence**.

Note: Each barcode has a unique value.

Below that, you will find a little test database and the function `showCatalogue` that pretty-prints a catalogue with a nice column layout, in contrast to the cluttered output of the default `show` implementation.

- `showPrice :: Int -> String`

  Displays a price in pounds, with exactly two decimal places.

- `showItem :: (Barcode, Item) -> [String]`

  Displays a single item in the database, but returns each field separately, so that we can individually pad them later depending on which column they belong to.

- `pad :: Int -> String -> String`

  Fills a given string with spaces in the end to fill up a desired minimum length.

- `showCatalogue :: Catalogue -> String`

  Uses the previous functions to display the whole catalogue:

  1. iteratively convert all items in the database to strings
  2. compute the limits of each column (i.e. the longest string plus 3 spaces for clarity)
  3. pad all entries with appropriately many spaces, depending on previously computed limits
  4. add vertical line between the column headers and the items

Here is the output for the example `testDB`:

```
*Resit> putStrLn (showCatalogue testDB)
Key             Name                                Price
-----------------------------------------------------------
0265090316581   Talisker Single Malt Whisky         £29.00
0903900739533   Bagpipes of Glory                   £10.00
9780201342758   Hutton - Programming in Haskell     £35.00
0042400212509   Frying pan                          £7.50
```

Exercise 1

As a first task, we want to be able to display customer orders for items in our database. An `Order` is a list of barcodes along with the desired quantity for each item to be purchased. You have to define a function `showOrder :: Catalogue -> Order -> String` that, given a catalogue and an order, displays all ordered item names, along with their quantity, individual price, and calculated final price. In addition, we also need you to calculate an incurred 20% VAT cost on top of the final price, as well as keep a running sum of the total cost in the displayed table.

Hints:

- We strongly recommend building your solution on top of the given `showCatalogue`, generalising existing helper functions to work with ordered items instead. For instance, you should replace the `showItem` function with a new version `showItem' :: ((Name, Price, Quantity), Int, Int) -> [String]` that excludes the barcode but additionally displays the extra information we need in each line.
- Running sums can easily be implemented using `Prelude.scanl`.
- To do the VAT calculation you will need to perform float-point arithmetic; `Prelude.fromIntegral` will prove useful.

For the provided `testOrder`, the output should look like this:


```
*Resit> putStrLn (showOrder testDB testOrder)
Name                               Qty    Rate     Amount   Running Sum
------------------------------------------------------------------------
Talisker Single Malt Whisky        2      £29.00   £58.00   £58.00
Hutton - Programming in Haskell    1      £35.00   £35.00   £93.00
------------------------------------------------------------------------
VAT (20%)                                                   + £18.60
                                                            = £111.60
```


Exercise 2

Let's now define a generalisation of the `get` function. Define a function `getItems :: [Barcode] -> Catalogue -> [Item]` that, given a list of `Barcode`s and a `Catalogue`, returns a list of `Item`s.

Test your code for errors by typing:

```
*Resit> getItems ["0001","9780201342758","0003"] testDB
```

It should return a list containing only the item for the textbook:

```
[("Hutton - Programming in Haskell",3500)]
```

1.1 The real database

We will see how fast our implementation of keymaps in `KeymapList` is. First, we need to turn on the timekeeping feature of GHCi. Type this at the prompt:

```
*Resit> :set +s
```

GHCi will now give you time (and memory use) information for each expression you ask it to evaluate.

Your file `Resit.hs` contains a few functions that we haven't shown yet. First of all, it can read in the database file `database.csv`. You can do this by writing:

```
*Resit> db <- readDB
Done
(0.77 secs, 900,764,384 bytes)
```

The database is now loaded and assigned to the variable `db`, and will remain in the computer's memory until you reload your file.

The database is pretty large, so it's not a good idea to try to print it on the screen. But you can ask for the size:

```
*Resit> size db
104651
(0.02 secs, 66,144 bytes)
```

Another thing that is provided is the function `samples`. This takes an integer, and provides the given number of random barcodes from the database.

```
*Resit> ks <- samples 3 db
(0.08 secs, 47,847,400 bytes)
*Resit> ks
["0073999020083","0038793018971","0323900007253"]
(0.01 secs, 115,872 bytes)
```


Note: Calling `samples` saves the sampled keys in a local file called `keys.cache`. When the time comes to compare different keymap implementations, we will need to reload the REPL and therefore lose the access to the variable holding the sampled keys. Therefore, later we will call `loadKeys` instead of `samples` in order to retrieve the stored keys from the file system.

You can then use `getItems` to look up the items corresponding to the keys.


```
*Resit> getItems ks db
[("The Beatles",90),("SIDARI SPINACH LINGUINE",50),("VICKS NYQUIL LQ CAP COLD/FLU",80)]
(0.12 secs, 66,626,368 bytes)
```

Note: Printing a lot of items would mess up with our measurements, so we should use `force` on the retrieved elements instead of showing them:

```
*Resit> force (getItems ks db)
()
(0.19 secs, 89,954,720 bytes)
```

Exercise 3

You now have all the ingredients to do your benchmark, you should measure the time of the following commands:

1. reading the database
2. sampling 1000 random keys
3. retrieving all 1000 items with the samples barcodes

Note: We do not really care about the time it takes to sample the keys, as it includes writing to the file system. What we do care about though, is the time it takes to read the database into memory and how fast our queries are.

1.2 Keymaps as hash tables

While implementing keymaps as lists of key-value pairs is straight-forward, lookups are costly as you need to linearly traverse the list until you find the queried key. You will now define a different implementation, while exposing the same interface to the users of your keymap library.

In the file `KeymapTable.hs` you will find a different declaration of the `Keymap` data type, as well as the skeletons of (a subset of) the functions in `KeymapList.hs`.

We will represent that catalogue not as a list but as an *array*. Just as a keymap associates keys with values, an array associates indexes with elements. Whereas the keys of a keymap can be anything, the indexes in an array are sequential integers (or similar), and so it is easy to compute exactly where to find the element. Hence, whereas looking up a key in a keymap takes time linear in the size of the keymap ($O(n)$), indexing into an array takes constant time ($O(1)$).

A hash table builds a keymap on top of an array by using a function, called a hash function, to convert keys of an arbitrary type into indexes that are integers in a given range. For the hash table to have good performance, any arbitrary key should be equally likely to map to any permitted index into the table. This makes it rare to have a *collision*, when two keys have the same hash. However, collisions are not impossible, and to cope with that case at each location in the array we will store a `KeymapList`, which we have conveniently already defined. By keeping the number of collisions small the lookup time for each `KeymapList` can be kept small, and hence we can arrange for lookup in the `KeymapTable` to be constant rather than linear time.

First, we will need to import the array type that GHC provides in the `GHC.Arr` module:

```
import GHC.Arr
```


The type of arrays `Array i e` has indexes of type `i` and elements of type `e`. You should be able to complete the exam if you understand the following functions from the array library:

- `range :: Ix i => (i, i) -> [i]`

  The type class `Ix` specifies types suitable for use as indexes to an array. Given a pair of bounds (a lowest and highest index, paired together), the function `range` returns all indexes between the bounds.

- `array :: Ix i => (i, i) -> [(i, e)] -> Array i e`

  To construct an array with indexes of type `i`, you need to give the bounds of the array (i.e. lowest and highest index, paired together), as well as an association list (i.e. list of index-value pairs) to populate the array under construction.

- `accumArray :: Ix i => (e -> a -> e) -> e -> (i, i) -> [(i, a)] -> Array i e`

  Since we expect to have several hash collisions, we need to careful when giving the association list argument to `array`. Thankfully, there is a more convenient array constructor for exactly this use case, `accumArray`, which also expects an accumulating function that combines two different associates for the same index into one. (Notice the different type variables `a` and `e`.)

- `amap :: (a -> b) -> Array i a -> Array i b`

  Iteratively applying a function on all elements of an array, without changing each indexes in any way.

- `(!) :: Ix i => Array i e -> i -> e`

  Return the element of an array at the given index.

- `assocs :: Ix i => Array i e -> [(i, e)]`

  Convert an array to an association list (i.e. a list of index-value pairs).

- `elems :: Array i e -> [e]`
  Collect all the elements of an array in a list.

For more details, see https://hackage.haskell.org/package/base/docs/GHC-Arr.html.

Exercise 4

To understand the relation between `array` and `accumArray`, write a function

```
gather :: Ix i => (i,i) -> [(i,a)] -> [(i,[a])]
```

that gathers together all entries with the same index. For instance,

```
gather (0,3) [(1,"c"),(3,"b"),(1,"a")]
  == [(0,[]),(1,["c","a"]),(2,[]),(3,["b"])]
```

Hint: Use list comprehension(s).

You can compute `gather` by scanning the list of entries once for each index within the range of the bounds. Observe that the following defines an equivalent to `accumArray`.

```
accumArray' :: Ix i => (e -> a -> e) -> e -> (i,i) -> [(i,a)] -> Array i e
accumArray' f e bounds entries =
  array bounds [ (i, foldl f e xs) | (i,xs) <- gather bounds entries ]
```

However `accumArray` is much more efficient than `accumArray'`, because (unlike `gather`) it does not repeatedly scan the list of entries.

We will, in fact, re-use the `KeymapList` implementation for storing multiple entries with the same hash. To avoid name clashing, we have to use a *qualified import*, which allows us to reference all identifiers inside a module by prefixing them with a given module identifier:


```
import qualified KeymapList as L
```

Next, we define the type of hash values to be integer values, and define a typeclass for types with an associated hash function:


```
type Hash = Int

class Hashable a where
   hash :: a -> Hash
```


The type `Int` is suitable for indexing an array (i.e. it satisfies the constraint `Ix Int`).

We can (naively) hash a character by taking its byte representation:

```
instance Hashable Char where
  hash = ord
```

Exercise 5

In our particular case, we will need to hash strings. In order to do that, you need to provide a `Hashable` instance, re-using the character hashing shown above. Use the following formula to calculate the hash of a string s:

$$s_0 * 31^{n-1} + s_1 * 31^{n-2} + \ldots + s_{n-1}$$

where s_i denotes the (hash of the) character of s at index i, and n is the length of the string.

We generalise this technique to hash any list of hashable items, which is why the instance declaration is of the form: `instance Hashable a => Hashable [a] where ...`.

Our arrays will be of type `Array Hash (L.Keymap k v)`, mapping hashes to keymap lists. But in order to have efficient lookups, we need to restrict the range of the hash function, which we can accomplish by taking the hash result *modulo* a fixed maximum threshold `capacity`:

```
hashMod :: Hashable a => a -> Int -> Hash
hashMod x cap = hash x `mod` cap
```

We need to specify an appropriate capacity for a given database, since a very low number would defeat the purpose of hashing due to too many collisions. Therefore, a safe option is to simply take the capacity to be the total number of entries in the database. After having constructed a keymap, we will want to query it using the hash of a barcode, thus we also store the calculated capacity in our data structure. The datatype `Keymap` with keys `k` and values `v` now becomes:

```
data Keymap k v = HashTable (Array Hash (L.Keymap k v)) Int
```

Exercise 6

In this exercise, you will implement for hash tables

```
fromList :: (Eq k, Hashable k) => [(k, v)] -> Keymap k v
```

whose type now reflects the fact that the type of keys has to be hashable. For each of the three parts below, fill in the appropriately marked part of `KeymapTable.hs`.

(a) Construct the array with the corresponding dimensions that associates each hash with a list of key-value pairs.

Hint: We recommend `accumArray` over `array` for constructing the array, otherwise you will spend significantly more time reading the database into memory.

(b) Transform the `Array Hash [(k, v)]` from the previous step to the desired `Array Hash (L.Keymap k v)`. We can now construct the hash table by giving the corresponding arguments, as shown `KeymapTable.hs`.

Hint: There is no need to change the previous indexes, just the values.

Exercise 7

Then, proceed with defining the following functions from the keymap API:

(i) `get :: (Eq k, Hashable k) => k -> Keymap k v -> Maybe v`

(ii) `toList :: (Eq k, Hashable k) => Keymap k v -> [(k, v)]`

(iii) `size :: Eq k => Keymap k a -> Int`

Exercise 8

We can now compare our hash-table approach with our initial list implementation. Perform the same sequence of steps as in Exercise 2, but now loading the saved keys instead of sampling:

```
*Resit> :r
[2 of 3] Compiling KeymapTable    ( KeymapTable.hs, interpreted )
[3 of 3] Compiling Resit          ( Resit.hs, interpreted )
Ok, three modules loaded.
*Resit> db <- readDB
Done
(??? secs)
*Resit> ks <- loadKeys
(??? secs)
*Resit> force (getItems ks db)
()
(??? secs)
*ResitSol>
```

You may see greater overhead in reading the database in memory, but hopefully you got an order of magnitude faster results for querying!