The Barcode Reader
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 8

Banks, Heijltjes, Melkonian, Sannella, Scott, Vlassi-Pandi, Wadler

Week 9
due 4pm Tuesday 15 November 2022
tutorials on Thursday 17 and Friday 18 November 2022

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email kendal.reid@ed.ac.uk if you cannot join your assigned tutorial.

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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.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

Every time you go to the supermarket, you depend on a database that converts barcodes to item names and prices. In this tutorial we consider two different data structures for representing such data, and compare their retrieval times.

The Haskell files that come with this tutorial are `Tutorial8.hs`, `KeymapList.hs`, and `KeymapTree.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,
                    invariant, keys,
                    ...
                  )
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`.

```
data 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 a wrapper that prevents us from using normal list functions on keymaps. 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 *constraint* `Eq k` mean that whatever the type `k`, it must support an *equality* operation. 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.

- `invariant :: Eq k => Keymap k a -> Bool`

  Checks the invariant that all keys should be unique.

- `keys :: Keymap k a -> [k]`

  Returns all the keys in a `Keymap`.

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

  Gives the number of entries in a `Keymap`.

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

  Given a key, looks up the associated value in a `Keymap`. Keys are matched using `(==)`, which is why the constraint `Eq k` is needed. The value returned is of type `Maybe a`, rather than `a`, because a key might not occur in a `Keymap`. See pages 138–139 of the textbook for an explanation of the type `Maybe a`.

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

  Given a key and a value, 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. To be precise, it takes the old keymap and returns the new keymap with the key updated to the given value.

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

  Deletes an entry with the given key from a keymap. To be precise, it takes the old keymap and returns the new keymap with the entry for the given key deleted.

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

  Narrows a keymap to those values that satisfy a given predicate. To be precise, it takes the old keymap and returns the new keymap with any entry that doesn't satisfy the predicate deleted.

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

  Exports the keymap as a list.

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

  Builds the keymap from a list.

Now that we know what `KeymapList` is like, we can start working on the file `Tutorial8.hs`. Just below the top, you will find the declarations:

```
import KeymapList

type Barcode = String
type Product = String
type Unit    = String

type Item    = (Product,Unit)

type Catalogue = Keymap Barcode Item
```

Firstly, we are importing the `KeymapList` module. Next, there are the type aliases `Barcode`, `Product` and `Unit`, whose values are strings, and `Item` which is a pair `(Product,Unit)`. Finally, we are using the type alias `Catalogue` whose values are keymaps that associate a `Barcode` with an `Item`. Because of the invariant, no barcode occurs more than once in a catalogue.

Below that, you will find a little test database.

Exercise 1

Write a function

```
getItems :: [Barcode] -> Catalogue -> [Item]
```

that given a list of `Barcode` and a `Catalogue` returns a list of `Item`. For example

```
getItems ["0001","9780201342758","0003"] testDB ==
    [("Thompson - \"Haskell: The Craft of Functional Programming\"","Book")]
```

returns only one item, since the first and last barcodes don't appear in the database.

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:

```
*Tutorial8> :set +s
```

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

Your file `Tutorial8.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 large, so it's better not to print it. But you can ask for its size:

```
*Tutorial8> 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.

```
*Tutorial8> ks <- samples 3 db
(0.08 secs, 47,847,400 bytes)
*Tutorial8> ks
["0073999020083","0038793018971","0323900007253"]
(0.01 secs, 115,872 bytes)
```

Due to lazy evaluation, the work of finding the sample keys does not occur until you print the answer (although it is negligible in the case of sampling).

Note: Calling `samples` dumps the sampled keys in a local file called `keys.cache`. When the time comes (in Exercise 7) to compare different keymap implementations, we will need to reload and therefore lose the access to the variable holding the sampled keys. A function is provided called `loadKeys` that we may call to retrieve the stored keys from the file system.

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

```
*Tutorial8> getItems ks db
[("The Beatles",90),("SIDARI SPINACH LINGUINE",50),("VICKS NYQUIL LQ CAP COLD/FLU",80)]
(0.12 secs, 66,626,368 bytes)
```

Printing a lot of items would mess up our measurements, so we use `force` to look up the elements without showing them:

```
*Tutorial8> force (getItems ks db)
()
(0.19 secs, 89,954,720 bytes)
```

Exercise 2

Load the database, check its size, sample one thousand keys from the database, and record the time it takes to apply `getItems` to the keys.

```
*Tutorial8> db <- readDB
Done
(??? secs)
*Tutorial8> size db
???
(??? secs)
*Tutorial8> ks <- samples 1000 db
```

```
(??? secs)
*Tutorial8> force (getItems ks db)
()
(??? secs)
```

If the database was two times bigger, how would you expect the time to change?

1.2 Keymaps as trees

In this part of the exercise we will build a different implementation of keymaps, based on trees rather than lists. In the file `KeymapTree.hs` you will find a different declaration of the `Keymap` data type, as well as the skeletons of the same functions defined in `KeymapList.hs`.

In `KeymapTree` we will implement the same *application programming interface (API)* as we had in `KeymapList`, so from the outside they will look the same. However, internally they will differ, and so will their performance.

(Actually, there are a few tiny differences in the API. A new function `depth`, which computes the depth of a tree, is exported. And where `KeymapList` often has constraint `Eq`, `KeymapTree` has constraint `Ord`, because we now need to check the order of the keys rather than just whether they are equal.)

See Chapter 20 (pages 195–198) of the textbook for an explanation of the idea behind search trees. The data is stored in the nodes of a tree. The left branch of a node only stores data that is *smaller* than the data at the node itself, while the right branch stores data that is *larger*.

Look at the data type for `Keymap`. It is more complicated than before:

```
data Keymap k a = Leaf
                | Node k a (Keymap k a) (Keymap k a)
```

The data type again defines a keymap storing keys of type `k` and values of type `a`. In all functions that rely on the order of keys we will need the constraint `Ord k`. This means that keys are *ordered*, that is we can compare keys using the functions `(<)`, `(>)`, `(==)`, `(<=)`, `(>=)`, and `(/=)`.

The constructors for `Keymap` are `Leaf` and `Node`. The value of `Leaf` is just a leaf. It stores no data, like the empty list `[]`. The second one does all the work. It carries (in order):

- a key of type `k`
- the associated value of type `a`
- a subtree on the left, which is a tree of type `Keymap k a`
- a subtree on the right, also a tree of type `Keymap k a`

When building keymap, we want to make sure that the tree remains sorted. That is, for any node with a certain key, the keys in the left subtree should all be smaller than that key, and the keys in the right subtree should all be larger. This property is checked by the `invariant` function. To ensure the invariant always holds, we don't export the constructors of `KeymapTree`, so that only way to construct or deconstruct a value of the type is with functions defined in this module.

In `Tutorial8.hs` change the line:

```
import KeymapList
```

to

```
import KeymapTree
```

and load `Tutorial8` in GHCi.

Think of what the following expression should return:

```
size (Node "0001" "some item" Leaf Leaf)
```

If you try it out, what does it say?

Now, we will complete the functions in `KeymapTree.hs`.

Exercise 3

Define the function `toList`, which takes a tree and returns the corresponding list of (key, value) pairs. Design it so that the returned list is in ascending order by key. For example,

```
toList testTree == [(1,10),(2,20),(3,30),(4,40)]
```

Exercise 4

Take a look at the function `set`. The function defines a helper function `go` to do the recursion, to avoid repeating the variables `key` and `value` too often in the definition. Complete the definition of this function.

Exercise 5

Complete the function `get`. Remember that you should return a value of type `Maybe a`. For example

```
get 3 testTree == Just 30
get 6 testTree == Nothing
```

Use QuickCheck to verify `prop_set_get`.

Exercise 6

Write a function `fromList` to turn a list into a tree. You should use the function `set` to add each element to the tree. You can use recursion over the input list, or you could use the higher-order functions `foldr` and `uncurry`. For example,

```
fromList [(4,40),(3,30),(1,10),(2,20)] == testTree
```

Use QuickCheck to verify `prop_toList_fromList`.

Now we will evaluate the performance of `KeymapTree`. Save the file `KeymapTree.hs` and load `Tutorial8.hs`. We will redo the steps we took in Section 1.1, replacing the list representation of key maps by the tree representation.

Exercise 7

Load the database, check its size and depth, load the sample of one thousand keys from before, and record the time it takes to apply `getItems` to the keys.

```
*Tutorial8> db <- readDB
Done
(??? secs)
*Tutorial8> size db
???
(??? secs)
*Tutorial8> depth db
???
(??? secs)
*Tutorial8> ks <- loadKeys
(??? secs)
*Tutorial8> force (getItems ks db)
()
(??? secs)
```

If the database was two times bigger, how would you expect the time to change?

2 Optional material

Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

Most of the functions below can be easily defined by converting a tree to a list, operating on the list, and converting it back to a tree. We use trees rather than lists because we are concerned with efficiency, so that is not appropriate. However, you might write the functions twice, once directly on trees and once via lists, and then use QuickCheck to confirm that they yield the same result.

Exercise 8

Define two functions `filterLT` and `filterGT`, such that `filterLT k t` is the tree that results from retaining all elements in `t` whose key is less than `k`. For example:

```
keys (filterLT "0900000000000" testDB) == ["0042400212509","0265090316581"]
keys (filterLT "0" testDB) == []
```

The function `filterGT k t` is the tree that results from retaining all elements in `t` whose key is greater than `k`. For example:

```
keys (filterGT "0900000000000" testDB) == ["0903900739533","9780201342758"]
keys (filterGT "0" testDB) ==
          ["0042400212509","0265090316581","0903900739533","9780201342758"]
```

We will need these functions for the next part of the exercise.

Exercise 9

Define the function `merge`, which takes two trees and produces a single tree of all their elements. Write a suitable `quickCheck` test for your function.

Note: If both trees have values for some key, use the value from the first tree.

Exercise 10

Define the function `select`, which takes a predicate and a tree, and returns a new tree that contains only entries where the value satisfies the predicate.