The Barcode Reader
Informatics 1 – Functional Programming: Tutorial 6

Heijltjes, Wadler

Due: The tutorial of week 8 (12/13 Nov.)
Reading assignment: Chapters 15–16 (pp. 280–336)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

The barcode reader

In this tutorial 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 `tutorial6.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,
                    size,
                    ...
                  )
where
```

This declaration means that `KeymapList.hs` 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 Eq k => 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`.

There are some more details to this statement. The requirement `Eq k =>` means that a key `k` must have *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 expressions of that type—otherwise, we would have no way to identify the right key in the keymap.

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 constructor `K` is just a wrapper, that prevents us from using normal list functions on keymaps. This is precisely the idea: we want to restrict the things that one can do with a keymap, because it's easy to mess things up.

Now, let's look at the functions in the file. Note that all the types have the requirement `Eq k =>`, because this is required by the `Keymap` data type.

- `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 requirement `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 `tutorial6.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 Catalog = Keymap Barcode Item
```

Firstly, we are importing the `KeymapList` module. Next, there are type aliases `Barcode`, `Product` and `Unit`, which are strings, and `Item` which is a pair `(Product,Unit)`. Finally, we are using the type alias `Catalog` to refer to a `Keymap` that associates `Barcode`s with `Item`s.

Below that, you will find a little test database.

Exercises

1. Before we can work on the database, we need some way of viewing it. If you try `testDB` or `show testDB` on the GHCi prompt, you will find it refuses to print. Instead, try:

   ```
   *Main> toList testDB
   ```

   However, this looks rather cluttered. We will follow the exercise starting on page 108 of the book in making it look pretty.

   (a) Write a function `longestProduct` that finds the longest product name in a list of `(Barcode,Item)`-pairs.

   (b) Write a function `formatLine` that, given a number (the desired length of the product name) prints the barcode, product and unit information, separated by dots, as a single line. For example:

   ```
   *Main> formatLine 7 ("0001",("product","unit"))
   "0001...product...unit"
   *Main> formatLine 7 ("0002",("thing","unknown"))
   "0002...thing.....unknown"
   ```

   You may assume that the product name is never longer than the desired length for it.

   (c) Write a function `printCatalog` that pretty-prints a `Catalog`. You will need to use `toList` (from the `KeymapList` module). Test your function by writing at the prompt:

   ```
   *Main> putStr (showCatalog testDB)
   ```

2. Next, we will start using the `get` function. Firstly, try the following expression:

   ```
   *Main> get "9780201342758" testDB
   ```

   You will see that the desired answer is preceded by the constructor `Just`. Now, try:

   ```
   *Main> get "000" testDB
   ```

   What did you get? The data type `Maybe` is defined as follows:

   ```
   Maybe a = Nothing
           | Just a
   ```

   (a) When you apply your function `get` with a certain key to the test database, what is the type it returns? What are the possible values (hint: there are five)?

   (b) Write a function `getItems` that, given a list of `Barcode`s, returns a list of `Item`s. Test your code for errors by typing:

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

   It should return a list containing only the item for the textbook. For this definition you will probably need to write helper functions.

   (Hint: if you find an item, put it in a list of its own, if you don't find one, return an empty list; then piece all the lists back together.)

The real database

Your file `tutorial6.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 ordering:

```
*Main> makeDB
```

(it might take a little while). The database has been loaded up as `theDB :: Catalog`, 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:

```
*Main> size theDB
```

Another thing that is provided is the function `getSample`. This will give you a random barcode from the database; try:

```
*Main> getSample
```

To find the `Item` for that barcode, follow the previous command with:

```
*Main> get it theDB
```

('`it`' is a GHCi-special that refers to the value returned by the last expression it evaluated).

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:

```
*Main> :set +s
```

It may seem as if nothing has changed, but GHCi will now give you time (and memory use) information for each expression you ask it to evaluate.

Exercises

3. (a) Reload your file and load up the database again. Take a note of how much time it took.
   (b) Take at least ten samples from the database, and record how much time it takes to find an item with `get`. (The time it takes to find a random sample is not really relevant here.)
   (c) Think about the different values you get. If the database was 2 times bigger, how much longer would it take (on average) to find an item? How many items does the `get` function from `KeymapList` look at before it finds the right item, if it happens to be the last one?

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 functions as in `KeymapList.hs` (plus two extra functions, `depth` and `merge`).

In `KeymapTree` we will implement the same functions and data type as we had in `KeymapList`, so from the outside they will look the same. However, internally they will be very different, and so will their performance.

The idea behind the tree implementation is explained in section 16.7 (page 315) of the textbook. Basically, 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*.

First, look at the data type for `Keymap`. It is a lot more complicated than before:

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

The data type again defines a keymap storing keys of type `k` and values of type `a`. To sort the keys into larger and smaller ones, we need the requirement `Ord k =>`. This means that the keys used in a keymap can be *ordered*; in practice, the means we can always use the functions `(==)`, `(>=)` and `(<=)` on keys.

The constructors for `Keymap`s then are

```
Node k a (Keymap k a) (Keymap k a)
Leaf
```

The second is a leaf; like the empty list `[]` it stores no data. The first 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 a keymap in the shape of a tree, 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 that that key, and the keys in the right subtree should all be larger. To ensure this, we make sure a user of these keymaps can only access them through functions that are safe.

Exercises

4. In `tutorial6.hs` change the line:

   ```
   import KeymapList
   ```

   to

   ```
   import KeymapTree
   ```

   and load `tutorial6.hs` up in GHCi. Think of what the following expression should return:

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

   If you try it out, what does it say?

What happens is that by not exporting the constructors `Node` and `Leaf` themselves, we prevent people from writing recursive functions on our trees—at least outside of `KeymapTree.hs`. Now, we will complete the functions in `KeymapTree.hs`.

Exercises

5. (a) Look at the function `size`. How does it work, and how can we recurse over trees?
   (b) Define the function `depth`, which takes a tree and returns the *maximal* depth of the tree, i.e. the length of the longest path from its root to any of its leaves. A leaf should have depth 0.
   (c) Load up `KeymapTree.hs` and into GHCi and try the functions `size` and `depth` on the little test tree `testTree` (it should have size 4 and depth 3).
6. Define the function `toList`, which takes a tree and returns the corresponding list of pairs. Try it on `testTree`. Can you make it so that the returned list is sorted?
7. Take a look at the function `set`. The function defines a helper function `f` to do the recursion, to avoid repeating the variables `key` and `value` too often in the definition.

   (a) Explain what the function `f` does when it encounters a leaf.

(b) Explain what the function `f` does when it looks at a node. What happens if it has found the key it was looking for? And if not, where does it continue to search?

8. Complete the function `get`. Remember that you should return a `Maybe`-value. When should it return `Nothing`, and when should it return `Just` a value? Test your function on `testTree` first, and then use QuickCheck to verify `prop_set_get`.
9. Write a function `fromList` to turn a list into a tree. You should use the function `set` to add each element to the tree. Think about what the tree is that you should start out with. For this question you can use recursion over the input list, but you could also try to use `foldr` and `uncurry`.

   Use the test property `prop_toList_fromList` to test your solutions. If you managed to return sorted lists with `toList`, you can also test with `prop_toList_fromList_sorted`.

At this point we have added enough functions to `KeymapTree` to start evaluating its performance.

Exercises

10. Save `KeymapTree.hs` and open up `tutorial6.hs`

    (a) Load up the database again by ordering `makeDB` at the prompt. This time, it will be constructed as a tree. How much time did it take?

    (b) Try on at least 10 examples how fast your `get` function is now. Remember:

    ```
    *Main> getSample
    ```

    gives you a random barcode, and then

    ```
    *Main> get it theDB
    ```

    looks up the associated item.

    (c) How many barcodes does our `get` function inspect, at most, when searching the database?

Optional material

Exercises

11. Look at the function `merge`. If you have not yet encountered the special symbol '`@`', can you guess what it does? Explain how the function `merge` works.
12. Define the function `del`, which takes a key and a tree, and returns a tree identical to its argument except with any entry for the given key deleted. You will need the function `merge` here.
13. 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.