Carrie's Candy Bowl Project

H. Conrad Cunningham

21 October 2018

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

Carrie's Candy Bowl Project

I wrote this assignment description for CSci 450/503 Assignment #3 in Fall 2018.

Problem Description and Initial Design

Carrie, the Department's Administrative Assistant, has a candy bowl on her desk. Often she fills this bowl with candy, but the contents are quickly consumed by students, professors, and staff members. In this project, we model the candy bowl.

At a particular point in time, the candy bowl may contain several different kinds of candy with zero or more pieces of each kind.

We can represent a candy bowl with the following user-defined Haskell algebraic data type `CandyBowl`:

```
data CandyBowl a = Bowl [a] deriving Show
```

In this definition, type parameter `a` denotes the type of the identifiers for the kinds of candy. For example, if we use strings for the kinds of candy, particular values might be `"Snickers"`, `"Kiss"`, and `"wintergreen mints"`.

In this representation, if the bowl contains two `"Snickers"` and nothing else, then the bowl would be represented by the following value:

```
Bowl ["Snickers","Snickers"]
```

We say that the bowl above contains no `"Kiss"` pieces.

Note: There are several possible representations for the candy bowl. We could use an association list (i.e. a list of pairs mapping kinds to counts) or an implementation of `Data.Map` to represent the candy bowl.

Exercises

Develop a Haskell module with the following functions using the type `CandyBowl` as defined above. You may use function you have completed to implement other functions in the list (as long as you do not introduce circular definitions).

Notes

- In some cases, you may need to restrict the polymorphism on `CandyBowl a` to implement a function. But be careful not to restrict functions unnecessarily.
- You may find Prelude functions such as `concatMap`, `elem`, `filter`, `length`, `map`, `null`, `replicate`, and `span` useful.
- You may also find functions in the `Data.List` library useful (e.g. `sort`, `group`, `(\\)`).

Functions

1. `newBowl :: CandyBowl a`
   creates a new empty candy bowl.
2. `isEmpty :: CandyBowl a -> Bool`
   returns `True` if and only if the bowl is empty.
3. `putIn :: CandyBowl a -> a -> CandyBowl a`
   adds one piece of candy of the given kind to the bowl.

   For example, if we use strings to represent the kinds, then

   ```
   putIn bowl "Kiss"
   ```

   adds one piece of candy of kind `"Kiss"` to the `bowl`.

4. `has :: CandyBowl a -> a -> Bool`
   returns `True` if and only if one or more pieces of the given kind of candy is in the bowl.

5. `size :: CandyBowl a -> Int`
   returns the total number of pieces of candy in the bowl (regardless of kind).

6. `howMany :: CandyBowl a -> a -> Int`
   returns the count of the given kind of candy in the bowl.

7. `takeOut :: CandyBowl a -> a -> Maybe (CandyBowl a)`
   attempts to remove one piece of candy of the given kind from the bowl (so it can be eaten). If the bowl contains a piece of the given kind, the function returns the value `Just bowl`, where `bowl` is the bowl with the piece removed. If the bowl does not contain such a piece, it returns the value `Nothing`

8. `eqBowl :: CandyBowl a -> CandyBowl a -> Bool`
   returns `True` if and only if the two bowls have the same contents (that is the same kinds of candy and the same number of pieces of each kind).

9. `inventory :: CandyBowl a -> [(a,Int)]`
   returns a Haskell list of pairs `(k,n)`, where each kind `k` of candy in the bowl occurs once in the list with `n > 0`. The list should be arranged in *ascending order* by kind.

   For example, if there are two `"Snickers"` and one `"Kiss"` in the bowl, the list returned would be `[("Kiss",1),("Snickers",2)]`.

10. `restock :: [(a,Int)] -> CandyBowl a`
    creates a new bowl such that for any `bowl`:

    ```
    eqBowl (restock (inventory bowl)) bowl  ==  True
    ```

11. `combine :: CandyBowl a -> CandyBowl a -> CandyBowl a`
    pours the two bowls together to form a new "larger" bowl.

12. `difference :: CandyBowl a -> CandyBowl a -> CandyBowl a`
    returns a bowl containing the pieces of candy in the first bowl that are not in the second bowl.

    For example, if the first bowl has four `"Snickers"` and the second has one `"Snickers"`, then the result will have three `"Snickers"`.

13. `rename :: CandyBowl a -> (a -> b) -> CandyBowl b`
    takes a bowl and a renaming function, applies the renaming function to all the kind values in the bowl, and returns the modified bowl.

    For example, for some mysterious reason, we might want to reverse the strings for the kind names: `f xs = reverse xs`. Thus `"Kiss"` would become `"ssiK"`. Then `rename f bowl` would do the reversing of all the names.