CS 312 Recitation 7
Stacks and queues, exceptions

Some functional data structures

In this recitation, we look at examples of structures and signatures that implement data structures. We show that stacks and queues can be implemented efficiently in a functional style.

What is a functional stack, or a functional queue? It is a data structure for which the operations do not *change* the data structure, but rather create a new data structure, with the appropriate modifications, instead of changing it in-place. In imperative languages, data operations generally support **destructive update** — "destructive" in the sense that after the update is done, the original data structure is gone. Functional abstractions support **nondestructive updates**: the original value is still around, unmodified, so code that was using it is unaffected. For efficiency, it is important to implement nondestructive updates not by creating an entirely new data structure, but by sharing as much as possible with the original data structure.

Recall a stack: a last-in first-out (LIFO) queue. Just like lists, the stack operations fundamentally do not care about the type of the values stored, so it is a naturally polymorphic data structure.

Here is a possible signature for functional stacks:

This signature specifies a parameterized abstract type for stack. Notice the type variable 'a. The signature also specifies the empty stack value, and functions to check if a stack is empty, and to perform push, pop and top operations on the stack. Moreover, we specify functions map and app to walk over the values of the stack.

We also declare an exception EmptyStack to be raised by top and pop operations when the stack is empty.

Here is the simplest implementation of stacks that matches the above signature. It is implemented in terms of lists.

```
structure Stack :> STACK =
  struct
    type 'a stack = 'a list
    exception Empty

    val empty : 'a stack = []
    fun isEmpty (l:'a list): bool =
      (case l of
         [] => true
       | _ => false)

    fun push (x:'a, l:'a stack):'a stack = x::l
    fun pop (l:'a stack):'a stack =
      (case l of
         [] => raise Empty
       | (x::xs) => xs)

    fun top (l:'a stack):'a =
      (case l of
         [] => raise Empty
       | (x::xs) => x)

    fun map (f:'a -> 'b) (l:'a stack):'b stack = List.map f l
    fun app (f:'a -> unit) (l:'a stack):unit = List.app f l
  end
```

Up until now, we have been defining exceptions solely in order to raise them and interrupt the executing program. Just like in Java, it is also possible to catch exceptions, which is termed 'handling an exception' in SML.

As an example, consider the following example. In the above code, we have implemented top and pop respectively as functions that return the first element of the list and the rest of the list. SML already defines functions to do just that, namely **`hd`** and **`tl`** (for head and tail). The function hd takes a list as argument and returns the first element of the list, or raises the exception **`Empty`** if the list is empty. Similarly for **`tl`**. One would like to simply be able to write in **`Stack`**: **`fun top (l:'a stack):'a = hd (l) fun pop (l:'a stack):'a stack = tl (l)`**

However, if passed an empty stack, top and pop should raise the EmptyStack exception. As written above, the exception List.Empty would be raised. What we need to do is intercept (or handle) the exception, and raise the right one. Here's one way to do it:

```
fun top (l:'a stack):'a =
  hd (l) handle List.Empty => raise EmptyStack
fun pop (l:'a stack):'a stack =
  tl (l) handle List.Empty => raise EmptyStack
```

The syntax for handling exceptions is as follows: ***`e handle exn => e'`***

where *e* is the expression to evaluate, and if *e* raises an exception that matches *exn*, then expression *e'* is evaluated instead. The type of e and *e'* must be the same.

Let us write an example more interesting than stacks. After all, from the above, one can see that they are just lists. Consider the queue data structure, a first-in first-out data structure. Again, we consider functional queues. Here is a possible signature:

```
  signature QUEUE =
    sig
      type 'a queue
      exception EmptyQueue

      val empty : 'a queue
      val isEmpty : 'a queue -> bool

      val enqueue : ('a * 'a queue) -> 'a queue
      val dequeue : 'a queue -> 'a queue
      val front : 'a queue -> 'a

      val map : ('a -> 'b) -> 'a queue -> 'b queue
      val app : ('a -> unit) -> 'a queue -> unit
    end
```

The simplest possible implementation for queues is to represent a queue via two stacks: one stack A on which to enqueue elements, and one stack B from which to dequeue elements. When dequeuing, if stack B is empty, then we reverse stack A and consider it the new stack B. [[NOTE TO INSTRUCTOR: Use a picture or two to explain this]]

Here is an implementation for such queues. It uses the stack structure Stack, which is rebound to the name S inside the structure to avoid long identifier names. [[NOTE TO INSTRUCTOR: Focus on the aspects of the code that you find more interesting.]]

```
structure Queue :> QUEUE =
    struct

      structure S = Stack

      type 'a queue = ('a S.stack * 'a S.stack)
      exception EmptyQueue

      val empty : 'a queue = (S.empty, S.empty)
      fun isEmpty ((s1,s2):'a queue) =
        S.isEmpty (s1) andalso S.isEmpty (s2)

      fun enqueue (x:'a, (s1,s2):'a queue) : 'a queue =
        (S.push (x,s1), s2)

      fun rev (s:'a S.stack):'a S.stack = let
        fun loop (old:'a S.stack, new:'a S.stack):'a S.stack =
          if (S.isEmpty (old))
            then new
          else loop (S.pop (old), S.push (S.top (old),new))
      in
        loop (s,S.empty)
      end

      fun dequeue ((s1,s2):'a queue) : 'a queue =
```

```
    if (S.isEmpty (s2))
      then (S.empty, S.pop (rev (s1)))
                handle S.EmptyStack => raise EmptyQueue
    else (s1,S.pop (s2))

  fun front ((s1,s2):'a queue):'a =
    if (S.isEmpty (s2))
      then S.top (rev (s1))
                handle S.EmptyStack => raise EmptyQueue
    else S.top (s2)

  fun map (f:'a -> 'b) ((s1,s2):'a queue):'b queue =
    (S.map f s1, S.map f s2)

  fun app (f:'a -> unit) ((s1,s2):'a queue):unit =
    (S.app f s2;
     S.app f (rev (s1)))

end
```

Folding

Over the past few weeks, we have seen many examples of data structures such as lists, dictionaries, stacks and queues which keep an arbitrary number of values in some kind of order. For all of those structures, we have (or can) define functions 'map' and 'app' to walk over the structures. For example, we have seen the functions:

```
List.map : ('a -> 'b) -> 'a list -> 'b list
Stack.map : ('a -> 'b) -> 'a stack -> 'b stack
Queue.map : ('a -> 'b) -> 'a queue -> 'b queue
```

These functions all apply a function (called a transformation function) of type 'a -> 'b over the list/stack/queue elements. For lists, for example, **`map f [x1,...,xn]`** produces a list **`[f (x1), ..., f (xn)]`**. The class of functions 'app' apply a function of type 'a -> unit for its side effects to all the elements of the data structure, in some specific order.

It turns out that such data structures often support a much more general operation called ``folding'', from which many higher-order functions can be derived automatically, including map and app, but also functions such as filter.

Folding is actually supported through a pair of operations, **`foldl`** and **`foldr`** (which stand for fold-left and fold-right). They all have type:

```
(('a * 'b) -> 'b) -> 'b -> 'a list -> 'b (for lists)
(('a * 'b) -> 'b) -> 'b -> 'a stack -> 'b (for stacks)
(('a * 'b) -> 'b) -> 'b -> 'a queue -> 'b (for queues)
```

We only discuss folding for lists in this recitation, but it should be clear that whatever we say for lists can be said for stacks, queues, or any other structure which supports folding.

Intuitively, folding applies a given operation to the elements of a list. Given a function f of type ('a * 'b) -> 'b, the expression

```
foldr f b [x1,x2,...,xn]
```

evaluates to

```
f (x1,f (x2,f (...,f(xn,b))))
```

and

```
foldl f b [x1,x2,...,xn]
```

evaluates to

```
f (xn,f (..., f (x2,f (x1,b))))
```

Here are the implementations: [NOTE TO INSTRUCTOR: it may be worthwhile to point out the differences between foldl and foldr in the code, as it relates to the above]]

```
fun foldl (f: ('a * 'b) -> 'b) (base : 'b) (l: 'a list) : 'b =
  case l of
    [] => b
  | x::xs => foldl f (f (x,base)) xs

fun foldr (f: ('a * 'b) -> 'b) (base : 'b) (l: 'a list): 'b =
  case l of
    [] => b
  | x::xs => f (x,foldr f base xs)
```

A lot of natural operations on lists can be implemented using folding. For example, getting the length of a list can be written as

```
fun length (l:'a list):int =
  foldl (fn (x:'a,len:int) => 1+len) 0 l
```

Similarly, computing the sum of a list of integers can be written

```
fun sum (l:int list):int =
  foldl (fn (x:int,sum:int) => x+sum) 0 l
```

Actually, in the above, both a foldl and a foldr can be used, since the operation + is commutative (!). We can implement map using folding (notice the foldr!):

```
fun map (f:'a -> 'b) (l:'a list) :'b list =
  foldr (fn (x:'a,r:'b list) => f (x)::r) [] l
```

and app as well:

```
fun app (f:'a -> unit) (l:'a list): unit =
  foldl (fn (x:'a,_:unit) => f (x)) () l
```

Filtering a list, that is keeping all the elements of a list for which a given predicate function returns true, can also be easily
implemented:

```
fun filter (p:'a -> bool) (l:'a list):'a list =
  foldr (fn (x:'a,r:'a list) => if p (x) then x::r else r) [] l
```

Even reversing a list can be implemented using a simple fold:

```
fun rev (l:'a list):'a list =
  foldl (fn (x:'a,xs:'a list) => x::xs) [] l
```

(which can be simply written as 'foldl (op ::) [] l')

Folding can be used to derive most higher-order functions that are often used. For many data structures, it is sufficient to provide folding functions over the structure to be able to derive functions such as map, app, filter, and others. In a precise sense, folding functions act as ``iterators'' over the elements of the data structure.

In a world where one programs with higher-order functions and folding, curried functions become more and more important. Consider the function Int.fmt, of type StringCvt.radix -> int -> string, which converts an integer to a string in the given radix representation (decimal, octal, binary, hexadecimal). Suppose we have a non-curried version of the function:

```
fun fmtNC (r:StringCvt.radix,i:int):string = Int.fmt r i
```

Suppose we wanted to convert a list L of integers to binary with such a function. Then we would use a map function as follows:

```
map (fn (x:int) => fmtNC (StringCvt.BIN,x)) L
```

whereas using the curried form, we get the more satisfying:

```
map (Int.fmt StringCvt.BIN) L
```

Notice how we do not have to define a new anonymous function in the second example, as it is automatically created by the currying process. As another example, consider the function String.isPrefix, of type string -> string -> bool. The call String.isPrefix s1 s2 returns true if s1 is a prefix of s2, and false otherwise. For the sake of arguments, suppose we have a non-curried version:

```
fun isPrefixNC (s1:string,s2:string):bool = String.isPrefix s1 s2
```

Suppose we have a list L of URLs from which we want to keep only the ftp sites (starting with ftp://), using the filter function above. If we want to use the non-curried function, we have to write:

```
filter (fn (x:string) => isPrefixNC ("ftp://",x)) L
```

whereas with the curried function, the form is much simpler:

```
filter (String.isPrefix "ftp://") L
```

In summary, using curried functions with higher-order functions often saves us from having to introduce spurious 'fn' to construct functions that simply perform bookkeeping on the arguments.