L28: Advanced functional programming

Exercise 2

Due on 8th March 2017

Submission instructions

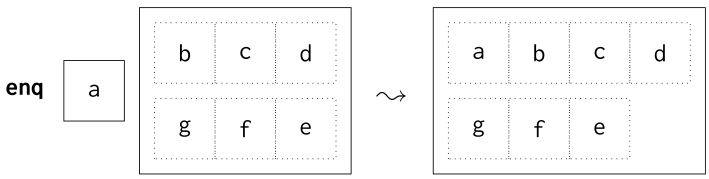
Your solutions for this exericse should be handed in to the Graduate Education Office by 4pm on the due date. Additionally, please email the completed text file `exercise2.ml` to `jeremy.yallop@cl.cam.ac.uk`.

1 Queues and length invariants

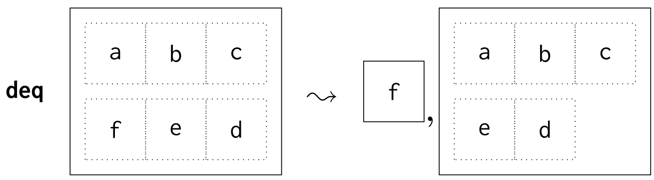
A *queue* is a type of sequence supporting two operations: `enq` adds an element to the front of the queue, and `deq` removes an element from the back. Besides `enq` and `deq`, queues also support operations for creating an empty queue and checking whether a queue is empty. Here is an OCaml signature for queues:

```
module type QUEUE = sig
  type +'a queue

  exception EMPTY

  val enq : 'a * 'a queue -> 'a queue
  (** insert an element at the front *)

  val deq : 'a queue -> 'a * 'a queue
  (** remove & return back element; raise EMPTY if queue is empty *)

  val empty : 'a queue
  (** an empty queue *)

  val is_empty : 'a queue -> bool
  (** whether the queue is empty *)
end
```

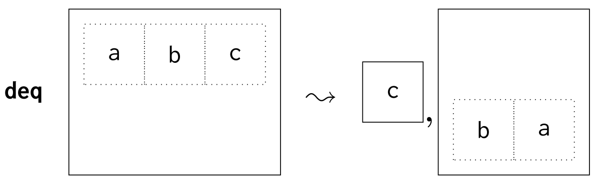
Representing a queue as a pair of lists, `inq` and `outq`, allows the operations to be implemented fairly efficiently. The `enq` operation *cons*es an element onto `inq`:



and `deq` removes the first element from `outq`:



If `outq` is empty, then `deq` first moves the elements of `inq` to `outq`, reversing their order:



Several invariants govern the behaviour of queues — for example, the element removed by `deq` must be the element least recently added by `enq` — and many of these invariants can be expressed using OCaml's types. This exercise focuses on invariants related to the length of the queue: `enq` increments the length, `deq` decrements the length, and the length of the queue is equal to the sum of the elements of `inq` and `outq`.

(*a*) Give an implementation of queues that satisifes the `QUEUE` signature using the following type definition:

```
type 'a queue = { inq: 'a list; outq: 'a list }
```

(3 marks)

(*b*) The following type represents the addition relation between three natural numbers (such as the lengths of `inq`, `outq`, and the queue formed by combining the two):

```
type (_,_,_) add =
    AddZ : 'n nat -> (z, 'n, 'n) add
  | AddS : ('m, 'n, 'o) add -> ('m s, 'n, 'o s) add
```

The two constructors represent two facts about addition: the type of `AddZ` says that if `n` is a `natural` number then `0 + n = n`, and the type of `AddS` says that if `m + n = o` then `succ(m)+ n = succ(o)`. The type `nat`, used in the definition of `AddZ` is defined as follows:

```
type _ nat =
    Z : z nat
  | S : 'n nat -> 'n s nat
```

This question asks you to define various additional functions representing further facts about addition that may be useful in the questions that follow.

(*i*) Define a function `jiggle` whose type says that if `m + succ(n)= o` then `succ(m)+ n = o`.

(*ii*) The type of `AddZ` states that for any `nat n`, `0 + n = n`, i.e. that zero is a left unit for addition. Define a function `addzr` whose type shows that zero is also a right unit for addition, i.e. that for any `nat n`, `n + 0 = n`.

(*iii*) Define a function `rz` that turns a proof that `m + 0 = n` into a proof that `m = n`.

(*iv*) Define two functions `addsr` and `inv_addsr` whose types show that `m + n = o` is equivalent to `m + succ(n)= succ(o)`.

(*v*) Define a type `commadd` whose type says that addition is commutative, i.e. that if `m + n = o` then `n + m = o`.

(6 marks)

(*c*) Here is a definition of vectors (length-indexed lists):

```
type ('a,'n) vec =
    Nil : ('a, z) vec
  | Cons : 'a * ('a, 'n) vec -> ('a, 'n s) vec
```

Define a function `rev` of the following type:

```
val rev : ('a, 'n) vec -> ('a, 'n) vec
```

such that `rev v` is the reverse of the vector `v`.

You may find it helpful to start by implementing the following function that catenates the reverse of one vector onto another:

```
val rev_append: ('a,'m) vec -> ('a,'n) vec -> ('m,'n,'o) add -> ('a,'o) vec
```

(3 marks)

(*d*) The following signature gives a more carefully-typed interface to queues:

```
module type TQUEUE = sig
  type ('a, 'n) queue
  val empty : (_, z) queue
  val isEmpty : (_, 'n) queue -> 'n isz
  val enq : 'a * ('a, 'n) queue -> ('a, 'n s) queue
  val deq : ('a, 'n s) queue -> 'a * ('a, 'n) queue
end
```

where the type `isz` indicates whether a natural number is zero:

```
type _ isz = IsZ : z isz | IsS : _ s isz
```

Starting from the following type definition, give an implementation of `TQUEUE`:

```
type ('a, 'n) queue =
  Queue : ('n, 'm, 'o) add * ('a, 'n) vec * ('a, 'm) vec ->
    ('a, 'o) queue
```

To receive full marks your implementation should return a valid value for every input, and should contain no unreachable code (i.e. `assert false` or similar).

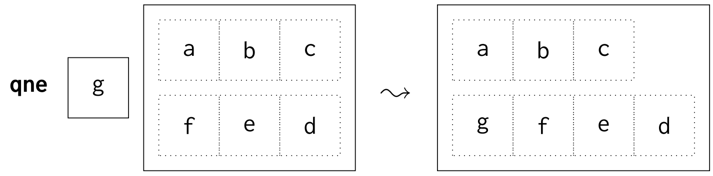
(5 marks)

(*e*) Complete the implementation of the following functor that builds an implementation of QUEUE from an implementation of TQUEUE.

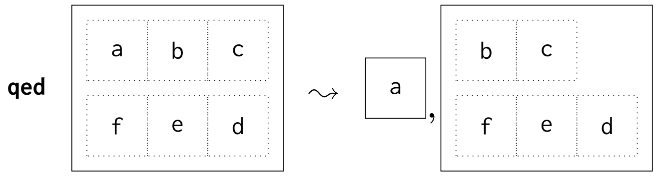
```
module Queue_of_tqueue(T: TQUEUE) : QUEUE = (* ... *)
```

(3 marks)

(*f*) A *deque* is a generalisation of a queue that support two additional operations: qne adds an element to the rear of the queue,



and qed removes an element from the front:



Starting from your implementation of TQUEUE, give an implementation of the following interface to deques:

```
module type TDEQUE =
sig
  include TQUEUE

  val qne : ('a, 'n) queue * 'a -> ('a, 'n s) queue
  val qed : ('a, 'n s) queue -> 'a * ('a, 'n) queue
end
```

(3 marks)

2 Sorted data structures

Lectures 8 and 9 showed how GADTs can be used to describe and constrain the shape of data. However, many programs have additional constraints on data that go beyond restrictions on shape. For example, it is often useful to order the elements in a sequence or a tree according to a user-defined function.

The `Set` module in the OCaml standard library is an example of parameterisation by order. The user of `Set` instantiates the functor `Set.Make` with a module `O` matching the `OrderedType` signature containing the type of elements and a function that compares two elements:

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

and `Set.Make` uses `O` to build an implementation of sets whose elements are stored in order.

However, nothing in the definition of `Map` ensures that the code used to implement ordered maps respects the order defined by `compare`. This exercise investigates how to make use of types to introduce ordering guarantees for an implementation of ordered trees, increasing confidence in the correctness of the code.

Changing the types to enforce ordering involves three steps.

First, we will associate a unique existential type variable with each element in the tree. Distinct elements `x` and `y` will be represented using distinct types `'x el` and `'y el`, and the types `'x` and `'y` will be used to represent elements in predicates. (Similar techniques appear in the presentation of *singletons* in Lecture 9 and in the presentation of *Lightweight Static Capabilities* in Lecture 7).

Second, we will introduce a type `le` to represent an ordering relation between variables. A value `('x,'y) le` serves as a proof of the fact that `'x el` is less than or equal to `'y el`. We will wrap the comparison function to return values of type `le`.

Finally, we will add suitable proofs (i.e. values of type `le`) to the constructors of the map, so that it is only possible to construct well-ordered maps.

The following module, `Typed_ordered`, will form the core of our implementation:

```
1  module Typed_ordered(O:ORDERED) : sig
2    type _ t
3    type ext = E : _ t -> ext
4    val inj : O.t -> ext
5
6    type (_,_) le
7    val le_refl : ('a,'a) le
8    val le_trans : ('a,'b) le -> ('b,'c) le -> ('a,'c) le
9
10   type ('a,'b) compare_result =
11     LE : ('a,'b) le -> ('a,'b) compare_result
12   | EQ : ('a,'b) le * ('b,'a) le -> ('a,'b) compare_result
13   | GE : ('b,'a) le -> ('a,'b) compare_result
14   val compare : 'x t -> 'y t -> ('x,'y) compare_result
15 end
```

Lines 2–4 define a type `t`, a wrapper type `ext`, and an injection function `inj` that builds a value of type `t` (wrapped as a value of type `ext`) from a value of type `O.t`. The existential type in the definition of `ext` ensures that each call to `inj` returns a value of type `t` whose parameter type is distinct from every other type in the program.

Lines 6–8 define the ordering relation `le`, and two ways of forming proofs. The `le_refl` value expresses the fact that `le` is reflexive, i.e. that `x <= x` for any `x`. Similarly, the `le_trans` value expresses the fact that `le` is transitive.

Finally, lines 10–14 define a function `compare` and its return type. There are three possible outcomes to a call to `compare x y`:

- a proof that `x <= y` (represented by `LE`)
- a proof that `x <= y` and `y <= x` (represented by `EQ`)
- a proof that `y <= x` (represented by `GE`)

The parameter `O` satisfies the `ORDERED` signature, which is similar to `Map.OrderedType`, but uses a variant as the return type of `compare`:

```
type ord = LT | EQ | GT
module type ORDERED = sig
 type t
 val compare : t -> t -> ord
end
```

(*a*) As in question 1, we begin with a loosely-typed implementation of the data type. Complete the implementation of the following module

```
module Tree (O:ORDERED) : sig
 type t
 val empty : t
 val add : O.t -> t -> t
 val mem : O.t -> t -> bool
 val remove : O.t -> t -> t
end = struct
  type topt = t option
  and t = Tree : topt * O.t * topt -> t
  (* ... *)
```

so that `empty` is an empty tree, `add e t` adds an element `e` to a tree `t`, `mem e t` indicates whether `t` contains `e`, and `remove e t` removes `e` from `t`.

Furthermore, all the functions should maintain the elements in the tree in sorted order according to `O.compare`.

(3 marks)

(*b*) Here is the beginning of a more carefully-typed implementation of trees.

```
module TTree (O:ORDERED) = struct

  module T = Typed_ordered(O)
  type 'a el = 'a T.t
  open T

  type ('min,'max) tne =
    | Tree : ('min,'k,'lmax) t
           * ('lmax,'k) le
           * 'k el
           * ('k,'rmin) le
           * ('rmin,'k,'max) t -> ('min,'max) tne
  and ('min, 'k, 'max) t =
      NE : ('min,'max) tne -> ('min,'k,'max) t
    | E : ('min,'k) eq * ('k,'max) eq -> ('min,'k,'max) t
```

A value of type `('min, 'max) tne` is a non-empty tree whose minimum element is a value of type `'min el` and whose maximum element is a value of type `'max el`.

Along with the left and right subtrees and the element of type `'k el`, a `Tree` constructor stores proofs relating `'k` to `'lmax` (the greatest element in the left subtree), and to `'rmin` (the least element in the right subtree). Together, these proofs ensure that only well-ordered trees can be constructed.

A value of type `('min, 'k, 'max) t` is a possibly-empty tree. There are two constructors: `NE` represents a non-empty tree, and `E` an empty tree. Empty trees carry proofs that `'min`, `'max` and `'k` are all equal, where equality is built from ordering:

```
type ('a, 'b) eq = ('a, 'b) le * ('b, 'a) le
```

(*i*) Continue the implementation of `TTree` by implementing a membership predicate with the following type:

```
val mem : 'k el -> ('min, 'max) tne -> bool
```

(3 marks)

(*ii*) Give an implementation of an insertion function `add` with the following type

```
val add : 'k el -> ('min,'max) tne ->
   ('k,'min,'max) add_result
```

where `add_result` is defined as follows:

```
type ('k, 'min, 'max) add_result =
| Least : ('k, 'min) le * ('k, 'max) tne ->
    ('k, 'min, 'max) add_result
| Greatest : ('max, 'k) le * ('min, 'k) tne ->
    ('k, 'min, 'max) add_result
| InRange : ('min, 'max) tne ->
    ('k, 'min, 'max) add_result
```

That is, there are three possibilities when inserting an element `k` into a tree with bounds ⟨`min`,`max`⟩:

- `k` is the least element (i.e. `k` ≤ `min`)
- `k` is the greatest element (i.e. `max` ≤ `k`)
- `k` lies between `max` and `min`

(3 marks)

(*iii*) Define an element removal function `remove` with the following type

```
val remove : 'k el -> ('min, 'max) tne ->
    ('k, 'min, 'max) remove_result
```

giving a suitable definition of `remove_result`. (Warning: this is quite tricky!)

(3 marks)