L28: Advanced functional programming

Exercise 2

Due on 8th March 2017

Submission instructions

Your solutions for this exercise 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: \texttt{enq} adds an element to the front of the queue, and \texttt{deq} removes an element from the back. Besides \texttt{enq} and \texttt{deq}, queues also support operations for creating an empty queue and checking whether a queue is empty. Here is an OCaml signature for queues:

\begin{verbatim}
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
\end{verbatim}

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

\begin{equation*}
\texttt{enq} \quad \begin{array}{c}
\begin{array}{c}
\hline
a \\
\hline
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\hline
\text{b} \\
\text{c} \\
\text{d} \\
\hline
\text{g} \\
\text{f} \\
\text{e}
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
\hline
a \\
\hline
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\hline
\text{b} \\
\text{c} \\
\text{d} \\
\hline
\text{g} \\
\text{f} \\
\text{e}
\end{array}
\end{array}
\end{equation*}

and \texttt{deq} removes the first element from \texttt{outq}:

\begin{equation*}
\texttt{deq} \quad \begin{array}{c}
\begin{array}{c}
\hline
\text{a} \\
\text{b} \\
\text{c} \\
\hline
\text{f} \\
\text{e} \\
\text{d}
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
\hline
\text{f}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\hline
\text{a} \\
\text{b} \\
\text{c} \\
\hline
\text{e} \\
\text{d}
\end{array}
\end{array}
\end{equation*}

If \texttt{outq} is empty, then \texttt{deq} first moves the elements of \texttt{inq} to \texttt{outq}, reversing their order:

\begin{equation*}
\texttt{deq} \quad \begin{array}{c}
\begin{array}{c}
\hline
\text{a} \\
\text{b} \\
\text{c}
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
\hline
\text{c}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\hline
\text{b} \\
\text{a}
\end{array}
\end{array}
\end{equation*}
Several invariants govern the behaviour of queues — for example, the element removed by \texttt{deq} must be the element least recently added by \texttt{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: \texttt{enq} increments the length, \texttt{deq} decrements the length, and the length of the queue is equal to the sum of the elements of \texttt{in} and \texttt{out}.

\textbf{(a)} Give an implementation of queues that satisfies the \texttt{QUEUE} signature using the following type definition:

\begin{verbatim}
    type 'a queue = { inq: 'a list; outq: 'a list }
\end{verbatim}

(3 marks)

\textbf{(b)} The following type represents the addition relation between three natural numbers (such as the lengths of \texttt{in}, \texttt{out}, and the queue formed by combining the two):

\begin{verbatim}
    type (_,_,_) add =
        AddZ: 'n nat -> (z, 'n, 'n) add
        | AddS: ('m, 'n, 'o) add -> ('m s, 'n, 'o s) add
\end{verbatim}

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

\begin{verbatim}
    type _ nat =
        Z: z nat
        | S: 'n nat -> 'n s nat
\end{verbatim}

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

\textbf{(i)} Define a function \texttt{jiggle} whose type says that if \( m + \text{succ}(n) = o \) then \( \text{succ}(m) + n = o \).

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

\textbf{(iii)} Define a function \texttt{rz} that turns a proof that \( m + 0 = n \) into a proof that \( m = n \).

\textbf{(iv)} Define two functions \texttt{addsr} and \texttt{inv_addsr} whose types show that \( m + n = o \) is equivalent to \( m + \text{succ}(n) = \text{succ}(o) \).

\textbf{(v)} Define a type \texttt{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):

\[
\text{type } ('a, 'n) \text{ vec} = \\
\quad \text{Nil : } ('a, z) \text{ vec} \\
\quad \text{Cons : } 'a \times ('a, 'n) \text{ vec} \to ('a, 'n s) \text{ vec}
\]

Define a function \( \text{rev} \) of the following type:

\[
\text{val } \text{rev} : ('a, 'n) \text{ vec} \to ('a, 'n) \text{ vec}
\]

such that \( \text{rev } v \) is the reverse of the vector \( v \).

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

\[
\text{val } \text{rev_append} : ('a, 'm) \text{ vec} \to ('a, 'n) \text{ vec} \to ('m, 'n, 'o) \text{ add} \to ('a, 'o) \text{ vec}
\]

(3 marks)

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

\[
\text{module type } \text{TQUEUE} = \text{sig} \\
\quad \text{type } ('a, 'n) \text{ queue} \\
\quad \text{val } \text{empty} : (_, z) \text{ queue} \\
\quad \text{val } \text{isEmpty} : (_, 'n) \text{ queue} \to 'n \text{ isz} \\
\quad \text{val } \text{enq} : 'a \times ('a, 'n) \text{ queue} \to ('a, 'n s) \text{ queue} \\
\quad \text{val } \text{deq} : ('a, 'n s) \text{ queue} \to 'a \times ('a, 'n) \text{ queue} \\
\end{sig}
\]

where the type \( \text{isz} \) indicates whether a natural number is zero:

\[
\text{type } _\text{isz} = \text{IsZ} : z \text{ isz} \mid \text{IsS} : s \text{ isz}
\]

Starting from the following type definition, give an implementation of \( \text{TQUEUE} \):

\[
\text{type } ('a, 'n) \text{ queue} = \\
\quad \text{Queue} : ('n, 'm, 'o) \text{ add} \times ('a, 'n) \text{ vec} \times ('a, 'm) \text{ vec} \to ('a, 'o) \text{ queue}
\]

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

(5 marks)

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

\[
\text{module Queue_of_tqueue(T: TQUEUE) : QUEUE = (* ... *)}
\]

(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,
\[
\begin{array}{c}
\text{qne} \\
g \quad a \quad b \quad c \\
f \quad e \quad d
\end{array} \quad \sim \quad
\begin{array}{c}
\text{a} \quad b \quad c \\
g \quad f \quad e \quad d
\end{array}
\]

and qed removes an element from the front:
\[
\begin{array}{c}
\text{qed} \\
\quad a \quad b \quad c \\
f \quad e \quad d
\end{array} \quad \sim \quad 
\begin{array}{c}
\text{a} \\
\quad b \quad c \\
f \quad e \quad d
\end{array}
\]

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

\[
\text{module type TDEQUE =}
\begin{align*}
\text{sig} \\
\text{include TQUEUE}
\end{align*}
\begin{align*}
\text{val qne : ('a, 'n) queue * 'a -> ('a, 'n s) queue} \\
\text{val qed : ('a, 'n s) queue -> 'a * ('a, 'n) queue}
\end{align*}
\]

(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:

```ocaml
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 e1 and ‘y e1, 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 e1 is less than or equal to ‘y e1. 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:
module Typed_ordered(O: ORDERED) : sig

  type _ t
  type ext = E : _ t -> ext
  val inj : O.t -> ext

  type (_,_) le
  val le_refl : ('a,'a) le
  val le_trans : ('a,'b) le -> ('b,'c) le -> ('a,'c) le

  type ('a,'b) compare_result =
    LE : ('a,'b) le -> ('a,'b) compare_result
    | EQ : ('a,'b) le * ('b,'a) le -> ('a,'b) compare_result
    | GE : ('b,'a) le -> ('a,'b) compare_result
  val compare : 'x t -> 'y t -> ('x,'y) compare_result

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 \( le_{\text{refl}} \) value expresses the fact that \( le \) is reflexive, i.e. that \( x \leq x \) for any \( x \). Similarly, the \( le_{\text{trans}} \) value expresses the fact that \( le \) is transitive.

Finally, lines 10–14 define a function \( \text{compare} \) and its return type. There are three possible outcomes to a call to \( \text{compare} \, x \, y \):  

- a proof that \( x \leq y \) (represented by \( \text{LE} \))
- a proof that \( x \leq y \) and \( y \leq x \) (represented by \( \text{EQ} \))
- a proof that \( y \leq x \) (represented by \( \text{GE} \))

The parameter \( O \) satisfies the \textit{ORDERED} signature, which is similar to \textit{Map.OrderedType}, but uses a variant as the return type of \( \text{compare} \):

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,'1max) t
        * ('1max,'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 '1max (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)