L28: Advanced functional programming

Exercise 2

Due date: see the course web page

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`.

Changelog

17:00 Sat 24th Feb	Added a note to question 3(e) clarifying the use of types to maintain queue invariants.

1 Alternative applicatives

The `APPLICATIVE` interface presented in Lecture 10 is the standard way of building applicative computations:

```
module type APPLICATIVE = sig
  type 'a t
  val pure : 'a -> 't
  val (<*>) : ('a -> 'b) t -> 'a t -> 'b t
end
```

However, there are several ways to define an interface equivalent to `APPLICATIVE`. For example, the following interface is based on a function `meld` that combines two computations by means of a function that combines their results:

```
module type MELDABLE = sig
  type 'a t
  val pure : 'a -> 'a t
  val meld : ('a  ->  'b  ->  'c) ->
             'a t -> 'b t -> 'c t
end
```

(We will make use of `MELDABLE` in Question 2.)

(*a*) Define functors with the following signatures that convert between `MELDABLE` and `APPLICATIVE`:

```
module Applicative_of_meldable(M: MELDABLE)
  : APPLICATIVE with type 'a t = 'a M.t = ...

module Meldable_of_applicative(A: APPLICATIVE)
  : MELDABLE with type 'a t = 'a A.t = ...
```

(*b*) Complete the proof of equivalence by defining a set of laws for the `MELDABLE` operations and showing that each set of laws follows from the other. You will need to give one proof for each of the `APPLICATIVE` laws and one proof for each of your new `MELDABLE` laws.

(Hint: you might start by introducing exactly the `MELDABLE` laws needed to complete the `APPLICATIVE` proofs.)

(10 marks)

2 Tries, sets and maps

In this section we will use the logarithm and exponentiation operations on types from Lecture 8 to construct a family of associative structures known as *tries*.

The tries we will build have several appealing features: they allow arbitrary data as keys, have purely functional behaviour (i.e. no mutation) and support an efficient lookup operation that is typically faster than the associative structures (hash tables and maps) in the OCaml standard library.

A trie with keys of type `k` and values of type `v` has a lookup function of type `k -> v`. It is possible to use this function type directly as a higher-order representation of tries. However, it is more convenient and efficient to use a first-order representation. The exponentiation laws from Lecture 8 give the rules for converting from a higher-order to a first-order representation.

Example: for a trie with keys of type `bool` and values of type `a`, the higher-order representation is

```
bool -> a
```

Interpreting `->` as exponentiation gives:

$$\mathtt{a}^{\mathtt{bool}}$$

and interpreting bool as a set with two inhabitants we have:

$$\mathtt{a}^2$$

which corresponds to a binary product:

$$\mathtt{a} \times \mathtt{a}$$

In general, the type of the keys determines the shape of the first-order trie representation. We'll write $\mathtt{trie}_k$ for the first-order representation corresponding to the key type `k`. The following three rules give the first-order representations for tries whose keys are built from units, sums and products:

key `k`	**higher-order trie $\mathtt{a}^k$**	$\mathtt{trie}_k$	**note**
`unit`	`unit -> a`	`a`	$\mathtt{a}^1 \equiv \mathtt{a}$
`s + t`	`s + t -> a`	$\mathtt{a\ trie}_s \ast \mathtt{a\ trie}_t$	$\mathtt{a}^{s+t} \equiv \mathtt{a}^s \times \mathtt{a}^t$
`s * t`	`s * t -> a`	$(\mathtt{a\ trie}_s)\ \mathtt{trie}_t$	$\mathtt{a}^{s\times t} \equiv (\mathtt{a}^s)^t$

Here is an interface to tries with keys `k` and representations `trie`:

```
module type TRIE =
sig
  type k
  type _ trie
  val all : 'v -> 'v trie
  val mix : ('a -> 'b -> 'c) -> 'a trie -> 'b trie -> 'c trie

  val set : k -> 'v -> 'v trie -> 'v trie
  val get : k -> 'v trie -> 'v
end
```

There are four functions:

- `all v` constructs a trie where every value is initially `v`.
- `mix f l r` combines the tries `l` and `r`, using `f` to combine the values for each key.
- `set k v t` updates the value for key `k` to `v` in the trie `t`. It returns a fresh copy of `t`, leaving the original unchanged.
- `get k t` returns the value corresponding to `k` in `t`. Since `t` stores a value for every possible `k`, `get` always succeeds.

The `all` and `mix` functions correspond to the `pure` and `meld` of the `MELDABLE` interface of Question 1, and so each trie can be treated as an `APPLICATIVE`.

Tries also follow a number of additional laws, such as (among others):

`get` retrieves the value stored by `all` when there are no intervening updates:

$$\texttt{get k (all v)} \quad \equiv \quad \texttt{v}$$

`get` retrieves the last value stored by `set` for the same key `k`:

$$\texttt{get k (set k v t)} \quad \equiv \quad \texttt{v}$$

Using `mix const` to combine a trie with itself has no effect:

$$\texttt{mix (fun x y -> x) t t} \quad \equiv \quad \texttt{t}$$

`mix f` followed by `get` is equivalent to `get` followed by `f`:

$$\texttt{get k (mix f t1 t2)} \quad \equiv \quad \texttt{f (get k t1) (get k t2)}$$

(a) (i) Following the rules in the table on page 4, write implementations of the `TRIE` interface for units, products and sums by supplying the parts marked `?` below:

```
implicit module Trie_unit :
  TRIE with type k = unit and type 'v trie = ?
 = ?

module Trie_product (A: TRIE) (B: TRIE)
  : TRIE with type k = A.k * B.k and type 'v trie = ? =
 = ?

module Trie_sum (A: TRIE) (B: TRIE)
 : TRIE with type k = (A.k, B.k) sum
        and type 'v trie = ?
  = ?
```

(*ii*) Many common types are isomorphic to combinations of units, sums and products. Rather than write a trie implementation for each type, we'll define a way to map those types into the implementations from question (*a*).

The `INJ` signature gives an interface to injections from a type `t` to a type `s`:

```
module type INJ = sig
  type t and s
  val inj : t -> s
end
```

Define a module `Trie_iso` that builds a trie implementation from an existing implementation `A` and an injection:

```
module Trie_iso (A: TRIE) (S: INJ with type s = A.k) :
  TRIE with type k = S.t and type 'v trie = 'v A.trie =
  ?
```

and use `Trie_iso` to build trie implementations with `bool` and `option` keys from `Trie_unit`, `Trie_product` and `Trie_sum`.

```
implicit module Trie_bool : TRIE with type k = bool
                                  and type 'v trie = ?
  = ?
implicit module Trie_option (A: TRIE) :
  TRIE with type k = A.k option and type 'v trie = ?
  = ?
```

(*iii*) The approach above works well for finite types, whose definitions don't involve recursion. However, types with an infinite number of inhabitants require a different approach.

The `default` type stores either a value or — if the value is not available — a default to be used in its place:

```
type ('p, 'a) default = Present : 'p -> ('p, 'a) default
                      | Absent  : 'a -> ('p, 'a) default
```

Using `default` we can define a `TRIE` implementation that uses less storage if most entries in the trie are unchanged from their initial value. Complete the following definition:

```
module Trie_default (A: TRIE) : TRIE
  with type k = A.k and type 'v trie = ('v A.trie,'v) default =
struct
  type k = A.k
  type 'v trie = ('v A.trie, 'v) default
  let all v = Absent v
  ...
```

The accompanying file `exercise2.ml` defines a functor `Trie_fix` that builds tries for recursively-defined types. If `F` is a functor that builds a trie with key type `b` from a trie with key type `a` then `Trie_fix` builds tries with key type μ`b.a`.

```
module Trie_fix (F:(functor (X: TRIE) -> TRIE)) :
sig
 module rec Fixed : sig
  type k = K : (F(Trie_default(Fixed)).k) -> k
  type 'v trie =  'v F(Trie_default(Fixed)).trie
  include TRIE with type k := k and type 'v trie := 'v trie
 end
end
```

Either using `Trie_fix` and `Trie_iso` **or** otherwise, give implementations of `TRIE` with `nat` and `list` keys:

```
implicit module Trie_nat
  : TRIE with type k = nat and type 'v trie = ?
 = ?
implicit module Trie_list {A: TRIE}
  : TRIE with type k = A.k list and type 'v trie = ?
 = ?
```

(If you choose not to use `Trie_fix`, your implementation should follow the approach underlying `Trie_sum`, `Trie_unit` and `Trie_product`.)

(*b*) The `TRIE` interface can be used to implement other collection interfaces, including `SET` and `MAP` as given below

```
module type SET = sig
  type t
  type elem

  (* Create an empty set *)
  val create : unit -> t

  (* Whether a set contains a particular element *)
  val member : elem -> t -> bool

  (* The union of two sets *)
```

```
  val mingle : t -> t -> t

  (* Add an element to a set *)
  val insert : elem -> t -> t

  (* Remove an element from a set *)
  val remove : elem -> t -> t
end

module type MAP = sig
  type 'v t
  type key

  (* Create an empty map *)
  val make : unit -> 'v t

  (* Retrieve the value corresponding to a key in a map *)
  val seek : key -> 'v t -> 'v option

  (* Join two maps together, using the function to merge values *)
  val join : ('a -> 'a -> 'a) -> 'a t -> 'a t -> 'a t

  (* Add a key to a map *)
  val push : key -> 'v -> 'v t -> 'v t

  (* Remove a key from a map *)
  val oust : key -> 'v t -> 'v t
end
```

Complete the following definitions to give implementations of `SET` and `MAP` based on `TRIE`. Each function in `Set` and `Map` should be implemented using the `TRIE` operations.

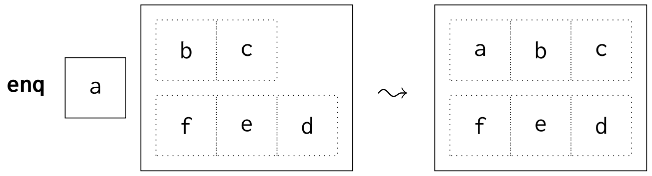
```
implicit module Set{T:TRIE} : SET with type elem = T.k = ?
implicit module Map{T:TRIE} : MAP with type key = T.k = ?
```

(12 marks)

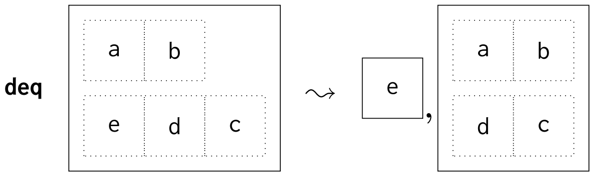
3 Queues and invariants

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

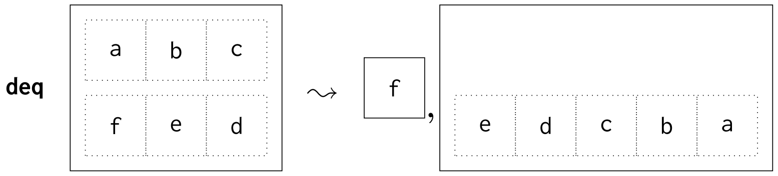
A common representing for queues in functional languages is a pair of lists, `inq` and `outq`. The `enq` operation *cons*es an element onto `inq`:



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



A common strategy to improve worst-case efficiency is to ensure that `inq` never grows longer than `outq`. Under this strategy, if `inq` grows longer than `outq`, then `enq` or `deq` additionally moves the elements of `inq` to `outq`, reversing their order:



Several other 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 the invariant relating the lengths of the lists that represent a queue.

Representing natural numbers

There are various ways to represent natural numbers, and some representations are better suited than others to particular tasks. In this exercise we'll use a representation based on differences, representing a number `o` as the difference between two other numbers `m` and `n`:

$$m - n = o$$

Among other properties, this representation ensures that $m \geq n$.

As with `max` in Lecture 7, we'll build representations of facts about numbers starting from primitive rules. Two facts about subtraction involving non-negative integers suffice:

$$\text{For any } n\text{:} \quad n - n = 0$$

$$\text{For any } m, n, o\text{:} \quad \text{if} \quad m - n = o \quad \text{then} \quad (m+1) - n = (o+1).$$

(a) Complete the following definition to give a type of proofs built from these facts:

```
type (_, _, _) sub =
    SubZ : (?, ?, ?) sub
  | SubS : (?, ?, ?) sub -> (?, ?, ?) sub
```

(b) The following fact is also useful, and may be derived:

$$\text{For any } m, n, o\text{:} \quad \text{if} \quad m - n = o \quad \text{then} \quad (m+1) - (n+1) = o.$$

Define a function `subsuc` that corresponds to a proof of the fact above:

```
val subsuc : (?, ?, ?) sub -> (?, ?, ?) sub
```

Length-indexed vectors

A *vector* is a kind of linked list that is indexed by its length, just as the trees in Lecture 8 were indexed by their depths.

Vectors may be defined with the following interface:

```
type (+'a,'n) vec

val nil : ('a, z) vec
val cons : 'a -> ('a, 'n) vec -> ('a, 'n s) vec
val length : ('a, 'n) vec -> ('n, z, 'n) sub
```

The type `vec` has two parameters representing the element type and the length. There are three functions

- `nil` constructs an empty vector (with length `z`).
- `cons` takes an element and a vector of length `n` and builds a vector whose length is the successor of `n`.
- `length` returns the length of a vector as a difference between natural numbers.

(*c*) Define a type of vectors `vec` along with functions `nil`, `cons` and `length` following the interface above.

(*Note*: either storing the length in the vector or recomputing the length on every call to `length` is acceptable.)

(*d*) Define a function `rev_append` along with a type `revappend_result` so that `rev_append v1 v2` appends the elements of `v2` in reverse order onto the tail of `v1`, and so that `revappend_result` represents the fact that the length of the vector returned by `rev_append` is equal to the sum of the lengths of its inputs:

```
type ('a, 'm, 'n) revappend_result

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

(*Hint*: we've been treating `sub` as a way of representing proofs about subtraction; it can also be seen as a representation of proofs about addition.)

(*e*) Implement the following interface to queues:

```
type 'a queue = Q : ('a, 'outlen) vec *
                    ('a, 'inlen) vec *
                    ('outlen, 'inlen, _) sub -> 'a queue

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

exception Empty
val deq : 'a queue -> 'a * 'a queue
```

so that `enq` and `deq` behave as defined above, `deq` additionally raises `Empty` iff the queue has no elements, and `isEmpty` returns `true` iff the queue has no elements.

(For full marks, your implementation should ensure using the types that `inq` cannot grow longer than `outq`.)

(13 marks)