Advanced Practice Sheet**, ***

Topics

- Functors
- Objects
- Polymorphic Variants

Exercise 1 – Abstract pairs and associative lists functors

Question 1.1 – First, write a functor `Pair` that takes two types `left_elt` and `right_elt` and returns a type `t` for pairs over these types.

Question 1.2 – Restrict the result of the `Pair` functor using a signature that makes `t` abstract. If necessary add functions so that pairs can be constructed and destructed from the outside.

Question 1.3 – Write a functor `Assoc`, whose result should be of the signature below, that takes an instance of `Pair`.

```
module type ASSOC_S = sig
  type t
  type key
  type value
  val empty : t
  val set : key -> value -> t -> t
  val get : key -> t -> value
  val unset : key -> t -> t
end
```

Exercise 2 – Extending functors

Question 2.1 – Write an `Extend_map` functor that takes an ordered type as parameter and returns a module of type `Map.S` extended with a `of_list` function. The type `Extend_map(M).t` should be equal to `Map.Make(M).t`

Question 2.2 – Write an `.mli` for that functor that does not hide any type information.

Question 2.3 – Continue extending that module with a `keys` function returning the set of bound keys in the map.

Exercise 3 – Lists wrapped in objects

Question 3.1 – Write an imperative list buffer object of the following type, whose insertion operations are all in $O(1)$. The call to `result ()` should be in $O(l)$ where l is the length of the result.

Successive calls to `result ()` should not take more time than one if no insertion has happened.

```
< append : 'a -> unit;
  prepend : 'a -> unit;
  concat_after : 'a list -> unit;
  concat_before : 'a list -> unit;
  result : unit -> 'a list >
```

Exercise 4 – Lists as objects

Question 4.1 – Write a virtual class `[ 'a ] list` that provides operations `hd` and `tl`, as well iterators `iter` and `fold`.

Question 4.2 – Derive a `[ 'a ] nil` class that implements the empty list.

Question 4.3 – Derive a `[ 'a ] cons` class that takes a head and a tail and prepends an element to an existing list.

Question 4.4 – Add a method `rev : 'a list`. You may add helper methods if needed.

Question 4.5 – Write a function `map : ('a -> 'b) -> 'a list -> 'b list` using the `iter` method. Write another one using the `fold` method.

Exercise 5 – Operator overloading using objects

In this exercise, we will define an addition operator (++) that can be overloaded with objects.

```
   let (++) x y = x#plus y
```

Question 5.1 – Declare the classes `cint` and `cfloat` such that the following code is valid.

```
# let a = new cint 1
# let b = new cint 2
# let c = a ++ b
# let s = c#show;;
val a : cint = <obj>
val b : cint = <obj>
val c : cint = <obj>
val s : string = "3"

# let a = new cfloat 1.
# let b = new cfloat 2.
# let c = a ++ b
# let s = c#show;;
val a : cfloat = <obj>
val b : cfloat = <obj>
val c : cfloat = <obj>
val s : string = "3."
```

To define binary methods, we need some method to access the internal representation of the classes.

Question 5.2 – Abstract the internal representation using a module signature.

Question 5.3 – Add a `float` method to the `cint` class returning a `cfloat`, and an floor method to the `cfloat` class returning a `cint`

Exercise 6 – Polymorphic variants rewriting

Given the following map definition

```
module StringMap = Map.Make(String)

type expr =
  [ `Plus of expr * expr
  | `Int of int
  | `Var of string ]

type expr_without_var =
  [ `Plus of expr_without_var * expr_without_var
  | `Int of int ]
```

Question 6.1 – Write `substitute` that rewrites `` `Var ``s into the value bound in the map. It should be of type

```
val substitute : [< expr ] StringMap.t -> [< expr ] -> [> expr_without_var]
```

For instance, v should be `` `Plus (`Int 1,`Int 2) ``

```
let v = subst (StringMap.singleton "x" (`Int 1)) (`Plus (`Var "x",`Int 2))
```

Question 6.2 – Write the `eval` function of type

```
val eval : [< expr_without_var] -> int
```

Advanced Practice Sheet

SOLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIONS

Solution to question 1.1

```
module type COMPONENT = sig
  type t
end

module Pair (L : COMPONENT) (R : COMPONENT) = struct
  type t = { left : L.t ; right : R.t }
  type left_elt = L.t
  type right_elt = R.t
  let make left right = { left ; right }
  let left { left } = left
  let right { right } = right
end
```

Solution to question 1.2

```
module type PAIR_S = sig
  type t
  type left_elt
  type right_elt
  val make : left_elt -> right_elt -> t
  val left : t -> left_elt
  val right : t -> right_elt
end

module Pair (L : COMPONENT) (R : COMPONENT)
  : PAIR_S with type left_elt = L.t
            and type right_elt = R.t = struct (* ... *) end
```

Solution to question 1.3

```
module type ASSOC_S = sig
  type t
  type key
  type value
  val empty : t
  val set : key -> value -> t -> t
  val get : key -> t -> value
  val unset : key -> t -> t
end

module Assoc (Pair : PAIR_S)
  : ASSOC_S with type key = Pair.left_elt
             and type value = Pair.right_elt = struct
  type t = Empty | Binding of Pair.t * t
```

```
  type key = Pair.left_elt
  type value = Pair.right_elt
  let empty = Empty
  let set k v l = Binding (Pair.make k v, l)
  let rec get k = function
    | Binding (pair, _) when Pair.left pair = k -> Pair.right pair
    | Binding (_, pairs) -> get k pairs
    | Empty -> raise Not_found
  let rec unset k = function
    | Binding (pair, pairs) when Pair.left pair = k -> unset k pairs
    | Binding (pair, pairs) -> Binding (pair, unset k pairs)
    | Empty -> Empty
end
```

Solution to question 2.1

```
module Extended_map(M:Map.OrderedType) : sig
  include Map.S with type 'a t = 'a Map.Make(M).t
               and type key = Map.Make(M).key
  val of_list : (key * 'a) list -> 'a t
end
= struct
  include Map.Make(M)
  let of_list l =
    List.fold_left (fun acc (k,v) -> add k v acc) empty l
end
```

Solution to question 2.2

```
module Extended_map(M:Map.OrderedType) : sig
  include Map.S with type 'a t = 'a Map.Make(M).t
               and type key = Map.Make(M).key
  val of_list : (key * 'a) list -> 'a t
end
```

Solution to question 2.3

```
module Extended_map(M:Map.OrderedType) : sig
  include Map.S with type 'a t = 'a Map.Make(M).t
               and type key = Map.Make(M).key
  val keys : 'a t -> Set.Make(M).t
end
= struct
  include Map.Make(M)
  module Set = Set.Make(M)
  let keys t =
    fold (fun k _ set -> Set.add k set) t Set.empty
end
```

Solution to question 3.1

```
let list_buffer () = object
  val mutable pre = []
  val mutable post = []
  method prepend x = pre <- [ x ] :: pre
  method append x = post <- [ x ] :: post
```

```
  method concat_before x = pre <- x :: pre
  method concat_after x = post <- x :: post
  method result () =
    let res =
      let open List in
      rev (fold_left
             (fun acc l -> rev_append l acc)
             [] (rev (rev_append (rev post) (rev pre)))) in
    pre <= [ res ] ; post <- [] ;
    res
end
```

Solution to exercise 4

```
class virtual [ 'a ] list = object
  method virtual hd : 'a
  method virtual tl : 'a list
  method virtual iter : ('a -> unit) -> unit
  method virtual fold : 'c. ('a -> 'c -> 'c) -> 'c -> 'c
  method virtual rev : 'a list
  method virtual rev_aux : 'a list -> 'a list
end

class [ 'a ] cons hd tl = object (self)
  inherit [ 'a ] list
  method hd = hd
  method tl = tl
  method iter f = f hd ; tl#iter f
  method fold f acc = tl#fold f (f hd acc)
  method rev = self#rev_aux (new nil)
  method rev_aux acc =
    tl#rev_aux (new cons hd acc)
end
and [ 'a ] nil = object (self)
  inherit [ 'a ] list
  method hd = failwith "hd"
  method tl = failwith "tl"
  method iter _ = ()
  method fold _ acc = acc
  method rev = (self :> 'a list)
  method rev_aux acc = acc
end

let map : ('a -> 'b) -> 'a list -> 'b list = fun f l ->
  let res = ref (new nil) in
  l#iter (fun v -> res := new cons (f v) !res) ;
  !res#rev

let map : ('a -> 'b) -> 'a list -> 'b list = fun f l ->
  (l#fold (fun v acc -> new cons (f v) acc) (new nil))#rev
```

Solution to exercise 5

```
class cint v = object (_:'self)
  val repr = v
```

```
  method repr = repr
  method plus (v:'self) = {< repr = v#repr + repr >}
  method show = string_of_int repr
end

class cfloat v = object (_:'self)
  val repr = v
  method repr = repr
  method plus (v:'self) = {< repr = v#repr +. repr >}
  method show = string_of_float repr
end
```

Solution to exercise 5

```
module Num : sig
  type repr_int
  type repr_float

  class cfloat : float ->
    object ('self)
      method repr : repr_float
      method plus : 'self -> 'self
      method show : string
    end

  class cint : int ->
    object ('self)
      method repr : repr_int
      method plus : 'self -> 'self
      method show : string
    end

end = struct
  type repr_int = int
  type repr_float = float

  class cint v = object (_:'self)
    val repr = v
    method repr = repr
    method plus (v:'self) = {< repr = v#repr + repr >}
    method show = string_of_int repr
  end

  class cfloat v = object (_:'self)
    val repr = v
    method repr = repr
    method plus (v:'self) = {< repr = v#repr +. repr >}
    method show = string_of_float repr
  end
end
```

Solution to exercise 5

```
module Num : sig
  type repr_int
```

```
  type repr_float

  class cfloat : float ->
    object ('self)
      method repr : repr_float
      method plus : 'self -> 'self
      method floor : cint
      method show : string
    end
  and cint : int ->
    object ('self)
      method repr : repr_int
      method plus : 'self -> 'self
      method float : cfloat
      method show : string
    end

end = struct
  type repr_int = int
  type repr_float = float

  class cint v = object (_:'self)
    val repr = v
    method repr = repr
    method plus (v:'self) = {< repr = v#repr + repr >}
    method float = new cfloat (float_of_int repr)
    method show = string_of_int repr
  end

  and cfloat v = object (_:'self)
    val repr = v
    method repr = repr
    method plus (v:'self) = {< repr = v#repr +. repr >}
    method floor = new cint (int_of_float repr)
    method show = string_of_float repr
  end
end
```

Solution to exercise 6

```
module StringMap = Map.Make(String)

type expr =
  [ `Plus of expr * expr
  | `Int of int
  | `Var of string ]

type expr_without_var =
  [ `Plus of expr_without_var * expr_without_var
  | `Int of int ]

let rec subst env (expr:[<expr]) : [> expr_without_var ] = match expr with
  | `Plus (e1, e2) -> `Plus (subst env e1, subst env e2)
  | `Var v -> subst env (StringMap.find v env)
```

```
  | `Int i as expr -> expr

let v = subst (StringMap.singleton "x" (`Int 1)) (`Plus (`Var "x",`Int 2))

let rec eval = function
  | `Plus (e1, e2) -> eval e1 + eval e2
  | `Int v -> v

let x = eval v
```