Problem Set 3 Solutions

Individual Problem [25]: Losing Your Marbles

There are many ways to solve this problem. Here we describe a few.

Let \( m \) be the number of marbles and \( c \) be the number of cups. Then one decomposition is as follows. If \( m \geq 1 \) and \( c > 1 \) then either (1) we put zero marbles in the first cup and distribute all the marbles among the remaining \( c - 1 \) cups or (2) we find all ways of distributing \( m - 1 \) marbles over all \( c \) cups and we place one extra marble in the first cup. This leads to the following code for the general case

\[
\text{(mapCons 0 (marbles m (c-1))) @ (mapIncHead (marbles (m-1) c))},
\]

where \text{mapCons} conses a given element to the front of every list in a list of lists and \text{mapIncHead} increments the head of every list in a list of lists. In this approach, there are two base cases:

1. If \( c = 1 \), all \( m \) marbles must go into the one cup; and
2. If \( m = 0 \), all \( c \) cups must be filled with 0 marbles.

Putting these cases together yields the following solution:

\[
\text{let rec marbles m c =}
\begin{align*}
\text{if } c = 1 \text{ then } [m] \\
\text{else if } m = 0 \text{ then allZeroes c} \\
\text{else (mapCons 0 (marbles m (c-1))) @ (mapIncHead (marbles (m-1) c))}
\end{align*}
\]

\[
\text{and allZeroes len =}
\begin{align*}
\text{if } len = 0 \text{ then } [] \\
\text{else 0 :: (allZeroes (len-1))}
\end{align*}
\]

\[
\text{and mapCons x yss =}
\begin{align*}
\text{match yss with} \\
\hspace{1cm} [] -> [] \\
\hspace{1cm} | (ys::yss') -> (x::ys)::(mapCons x yss')
\end{align*}
\]

\[
\text{and mapIncHead zss =}
\begin{align*}
\text{match zss with} \\
\hspace{1cm} [] -> [] \\
\hspace{1cm} | (zs::zss') -> (* Assume zs is non-empty *) \\
\hspace{1cm} \hspace{1cm} (((List.hd zs) + 1) :: (List.tl zs)) :: (mapIncHead zss')
\end{align*}
\]

The three auxiliary functions \text{allZeroes}, \text{mapCons}, and \text{mapIncHead} can all be written more succinctly with \text{map} and \text{range}:

\[
\begin{align*}
\text{and allZeroes len = ListUtils.map (fun x -> 0) (ListUtils.range 1 len)} \\
\text{and mapCons x yss = map (FunUtils.cons x) yss} \\
\text{and mapIncHead zss = map (fun zs -> ((List.hd zs) + 1) :: (List.tl zs)) zss}
\end{align*}
\]

The \text{allZeroes} case is actually superfluous, since it can be implemented by a recursive call to \text{marbles}:
let rec marbles m c =
  if c = 1 then [[m]]
  else if m = 0 then mapCons 0 (marbles m (c-1))
  else (mapCons 0 (marbles m (c-1))) @ (mapIncHead (marbles (m-1) c))

Another solution strategy is based on the following observation about the general case:

In the general case, we can put anywhere between 0 and \( m \) marbles in the first cup, and distribute the remaining marbles over the remaining \( c - 1 \) cups.

Expressing this in code leads to the following solution:

let rec marbles1 m c =
  if c = 1 then [[m]]
  else let rec combineMarbles i =
    if i > m then []
    else mapCons i (marbles1 (m - i) (c - 1)) @ (combineMarbles (i+1))
  in combineMarbles 0

In the local recursive function combineMarbles, the parameter \( i \) ranges over the integers between 0 and \( m \), inclusive, which are the number of marbles that go in the first cup. The number \( i \) is consed on to the front of all solutions for the remaining \( m-i \) marbles in \( c-1 \) cups. The lists for each \( i \) are appended together. Having \( i \) range from 0 up to \( m \) guarantees that the resulting lists will be in lexicographic order. Note that no base case for \( m = 0 \) is required since this case is correctly handled by the general case to yield a list with \( c \) zeroes.

This solution can be simplified using higher-order functions. Here is one such simplification:

let rec marbles m c =
  if c = 1 then [[m]]
  else ListUtils.flatten
    (ListUtils.map (fun i -> ListUtils.map (FunUtils.cons i)
      (marbles (m - i) (c - 1)))
     (ListUtils.range 0 m))

It is natural to use ListUtils.map in conjunction with ListUtils.range to encode the idiom of doing something for every integer between 0 and \( m \). The mapcons in the above solution is accomplished by invoking ListUtils.map with a partially applied FunUtils.cons. Since ListUtils.map in this case will return a list of lists of lists, we use ListUtils.flatten to flatten this into a list of lists. Note that the flatten and map can be combined using foldr:

let rec marbles m c =
  if c = 1 then [[m]]
  else ListUtils.foldr
    (fun i ans ->
     (ListUtils.map (FunUtils.cons i)
      (marbles (m - i) (c - 1)))
     @ ans)
    (ListUtils.range 0 m)
Group Problems

Group Problem 1 [15]: Substitution Model

a. [3] app5 (sub 1)
    ⇒ app5 ((fun x -> fun y -> x - y) 1)
    ⇒ app5 (fun y -> 1 - y)
    ⇒ (fun f -> f 5) (fun y -> 1 - y)
    ⇒ (fun y -> 1 - y) 5
    ⇒ 1 - 5
    ⇒ -4

Alternatively:

app5 (sub 1)
⇒ (fun f -> f 5) (sub 1)
⇒ sub 1 5
⇒ (fun x y -> x - y) 1 5
⇒ 1 - 5
⇒ -4

b. [4] app5 sub 2
    ⇒ (fun f -> f 5) sub 2
    ⇒ sub 5 2
    ⇒ (fun x y -> x - y) 5 2
    ⇒ 5 - 2
    ⇒ 3

c. [4] app5 (flip sub) 2
    ⇒ (fun f -> f 5) (flip sub) 3
    ⇒ flip sub 5 3
    ⇒ (fun f a b -> f b a) sub 5 3
    ⇒ sub 3 5
    ⇒ -2

d. [4] (flip app5) 4 sub
    ⇒ ((fun f -> fun a b -> f b a) app5) 4 sub
    ⇒ (fun a b -> app5 b a) 4 sub
    ⇒ app5 sub 4
    ⇒ (fun f -> f 5) sub 4
    ⇒ sub 5 4
    ⇒ (fun a b -> a - b) 5 4
    ⇒ 5 - 4
    ⇒ 1

Group Problem 2 [20]: Function Types

Figures 1 and 2 show each of the OCaml functions, along with the type given to them by the OCaml type reconstructor. Here are some notes on types repeated from the assignment:

- The particular type variable names used don’t matter, so any consistent renamings of the following types are also valid. For example, the type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b could also be written ('c -> 'e) -> ('q -> 'c) -> 'q -> 'e.
- Arrow types are right associative, so the type 'a -> 'b -> 'c -> 'd is parsed as if it were written 'a -> ('b -> ('c -> 'd))
- The product constructor * binds more tightly than ->, so 'a * 'b -> 'c * 'd means ('a * 'b) -> ('c * 'd) and not 'a * ('b -> 'c) * 'd
- A type variable (such as 'a or 'b) can be instantiated to any type in each use of a function with that type. For example, n_fold can be used with type int -> (int -> int) -> int -> int at one point of a program and int -> (bool -> bool) -> bool -> bool at another.
let id x = x
val id : 'a -> 'a

let compose f g x = (f (g x));
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

let rec repeated n f =  
    if (n = 0) then id else compose f (repeated (n - 1) f)
val repeated : int -> ('a -> 'a) -> 'a -> 'a

let uncurry f (a,b) = (f a b)
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

let curry f a b = f(a,b)
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

let chPair x y f = f x y
val chPair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c

let rec gen next isDone seed =  
    if (isDone seed) then []
    else
        seed :: (generate next isDone (next seed))
val gen : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list

let rec map f xs =  
    match xs with  
    | [] -> []  
    | (x::xs') -> (f x) :: (map f xs')
val map : ('a -> 'b) -> 'a list -> 'b list

let rec filter pred xs =  
    match xs with  
    | [] -> []  
    | (x::xs') ->  
        if (pred x) then x::(filter pred xs')  
        else filter pred xs'
val filter : ('a -> bool) -> 'a list -> 'a list

let product fs xs =  
    map (fun f -> map (fun x -> (f x)) xs) fs
val product : ('a -> 'b) list -> 'a list -> 'b list list

let rec zip pair =  
    match pair with  
    | ([], _) -> []  
    | (_, []) -> []  
    | (x::xs', y::ys') -> (x,y)::(zip(xs',ys'))
val zip : 'a list * 'b list -> ('a * 'b) list

Figure 1: Types of OCAML functions, part 1.
let rec unzip xys =  
match xys with  
  [] -> ([], [])  
| ((x,y)::xys') ->  
  let (xs,ys) = unzip xys'  
  in (x::xs, y::ys)  
val unzip : ('a * 'b) list -> 'a list * 'b list

let rec foldr binop init xs =  
match xs with  
  [] -> init  
| (x::xs) -> binop x (foldr binop init xs)  
val foldr : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b

let foldr2 ternop init xs ys =  
foldr (fun (x,y) ans -> (ternop x y ans)) init (zip(xs,ys))  
val foldr2 : ('a -> 'b -> 'c -> 'c) -> 'c -> 'a list -> 'b list -> 'c

let flatten xss = foldr (@) [] xss  
val flatten : 'a list list -> 'a list

let rec forall pred xs =  
match xs with  
  [] -> true  
| (x::xs') -> pred(x) && (forall pred xs')  
val forall : ('a -> bool) -> 'a list -> bool

let rec exists pred xs =  
match xs with  
  [] -> false  
| (x::xs') -> (pred x) || (exists pred xs')  
val exists : ('a -> bool) -> 'a list -> bool

let rec some pred xs =  
match xs with  
  [] -> None  
| (x::xs') -> if (pred x) then Some x else some pred xs'  
val some : ('a -> bool) -> 'a list -> 'a option

let oneListOpToTwoListOp f =  
let twoListOp binop xs ys = f binop (zip(xs,ys))  
in twoListOp  
oneListOpToTwoListOp : ('a -> ('b * 'c) list -> 'd) -> 'a -> 'b list -> 'c list -> 'd

let some2 pred = oneListOpToTwoListOp some pred  
val some2 : ('a * 'b -> bool) -> 'a list -> 'b list -> ('a * 'b) option

Figure 2: Types of OCAML functions, part 2.
Group Problem 3 [45]: Higher-Order List Functions

There are many valid solutions for these functions. Here are some of them.

a. [5] val sum_multiples_of_3_or_5 : int * int -> int
   let sum_multiples_of_3_or_5 m n = 
   foldr (+) 0 
   (filter (fun x -> (x mod 3) = 0 || (x mod 5) = 0) 
   (range m n))

b. [5] val all_contain_multiple : int -> int list list -> bool
   let all_contain_multiple n xss=
   for_all (fun xs -> (exists (fun x -> (x mod n) = 0)) xs) xss
   This can be simplified via two applications of eta-reduction to:
   let all_contain_multiple n = 
   for_all (exists (fun x -> (x mod n) = 0))

c. [5] val inner_product : int list -> int list -> int
   let inner_product xs ys = foldr (+) 0 (map2 ( * ) xs ys)

d. [5] val alts : 'a list -> 'a list * 'a list
   let alts = foldr (fun x (ys,zs) -> (x::zs, ys)) ([], [])

e. [5] val cartesian_product : 'a list -> 'b list -> ('a * 'b) list
   let cartesian_product xs ys = 
   flatten (map (fun x -> map (pair x) ys) xs)
   As in the marbles problem, the flatten/map combination can be expressed with a single foldr:
   let cartesian_product xs ys = 
   foldr (fun x ans -> (map (pair x) ys) @ ans) xs)

f. [5] val bits : int -> int list
   let bits n = 
   if n = 0 then 
   [0] 
   else 
   rev (map ((flip (mod)) 2) 
   (gen ((flip (/)) 2) ((=) 0) n))

g. [5] val n_fold : int -> ('a -> 'a) -> 'a -> 'a
   The following solution makes a list of n copies of the function f and then uses foldr to compose them all:
   let n_fold n f = foldr o id (map (fun _ -> f) (range 1 n))
   The idiom students typically find trickiest about the above solution is the mapping of the constant function fun _ -> f, which ignores its arguments and always returns f. Constant functions are handy in many situations. The foldr/map combination can be performed using a single foldr:
let n_fold n f = foldr (fun _ ans -> (o f ans)) id (range 1 n))

The map/range combination can be expressed via ana:

let n_fold n f =
    foldr o id (ana (fun i -> if i = 0 then None
        else Some (f, i-1))
        n)

Another solution strategy involves using iterate:

let n_fold n f =
    fst (iterate (fun (fcn,i) -> (o f fcn, i-1))
        (fun (_,i) -> i=0)
        (id,n))

h. [5] val inserts : 'a -> 'a list -> 'a list list

Here is a version using foldr':

let inserts x =
    foldr' (fun y ys' ans -> (x :: y :: ys') :: (map (cons y) ans))
        [[x]]

Here is an alternative version expressed in terms of foldr:

let inserts x =
    foldr (fun y ans ->
        match ans with
        (._ :: ys') :: _ -> (x :: y :: ys') :: (map (cons y) ans)
        | _ -> [] (* can’t happen; just makes type checker happy *)
        )
        [[x]]

i. [5] val permutations : 'a list -> 'a list list

let permutations xs = foldr (fun x ans -> flatten (map (inserts x) ans)) [[[]]] xs

Group Problem 4 [20]: Functional Sets

a. [12]: PredSet

A complete definition of PredSet is presented in Fig. 3. Since a returned set is a predicate, it can always be written in the form (fun y -> ...). With an explicit fun, the definition of fromPred would be

let fromPred p = fun y -> p y

but eta reduction simplifies this to

let fromPred p = p

Similar remarks hold for toPred. The reason that fromPred and toPred are necessary is that data abstraction requires that there be an explicit conversion between the concrete and abstract type, even when they happen to be the same.
module PredSet : PRED_SET = struct

  type 'a set = 'a -> bool

  let empty = fun y -> false

  let singleton x = fun y -> (x = y)
  (* Alternatively:
   
   let singleton x = ((=) x)

   *)

  let member x s = s x

  let union s1 s2 = fun y -> (s1 y) || (s2 y)

  let intersection s1 s2 = fun y -> (s1 y) && (s2 y)

  let difference s1 s2 = fun y -> (s1 y) && (not (s2 y))

  let fromList xs = fun y -> List.exists ((=) y) xs
  (* Alternatively:

   let fromList xs = foldr (fun x ans -> union (singleton x) ans) xs

   This can be written more succinctly as:

   let fromList xs = foldr (o union singleton) xs

   *)

  let fromPred p = p

  let toPred s = s

end

Figure 3: Implementation of PredSet.
b. [8]: Other Functions

The following answers underscore that the two approaches to representing sets have very different strengths and weaknesses.

1. The `toList` function *cannot* be added to the `PRED_SET` signature. In general, a predicate can specify an infinite set, and there is no way to represent such a set with a necessarily finite list.

2. The `fromPred` function *cannot* be added to the `SET` signature. In general, a predicate can specify an infinite set, and it would not be possible to implement the `toList` function for such a set.

3. It is easy to add `isEmpty` to any `SET` implementation. The following is a definition that works in any such implementation:

   ```ocaml
   let isEmpty s = (toList s = [])
   ```

   However, it is *not* possible to implement `isEmpty` for `PRED_SET`. The only way to tell if a given set-as-predicate is empty is to show that it returns `false` for every possible element of a given type. Since most OCAML types types have an infinite number of elements (at least conceptually), this is not possible. Being able to answer such a question is tantamount to the halting problem, which is known not to have any computable solution. Indeed, this is an instance of what is known as Rice’s Theorem: any non-trivial property of a set is undecidable.¹

4. In `PRED_SET`, `complement` is easy to define as:

   ```ocaml
   let complement s = fromPred (o not (toPred s))
   ```

   But `complement` is *not* possible to define in `SET`; the complement of a finite set would be infinite, requiring `toList` to return a list of infinite length.

5. The `isSubset` function *can* be implemented in a `SET` implementation by testing if every element of the first set is a member of the second. This is impossible in a `PRED_SET` implementation, because there is no way to enumerate all the elements of an arbitrary set (which could be infinite).

¹Both the halting problem and Rice’s Theorem are covered in CS235.