Precept 4: Proofs about Fuctional Programs

This precept will help familiarize you with material on proving things about programs. As part of your homework this week, you must read the online notes about proving things about programs. Refer to these notes to help you through this week's precept materials.

http://www.cs.princeton.edu/~dpw/courses/cos326-12/notes/reasoning.php

http://www.cs.princeton.edu/~dpw/courses/cos326-12/notes/reasoning-data.php

Part I

1. Consider the function tail:

```
let tail (xs: 'a list) : 'a list =
  match xs with hd :: tail -> tail
;;
```

Is tail a total function?

Is tail [] a valuable expression?

Is tail [3] a valuable expression?

2. Consider safediv:

```
let safediv (nums : int * int) : int option =
  let (x,y) = nums in
  if y == 0 then None
  else Some (x/y)
;;
```

Is safediv a total function?

Is safediv (1, 0) valuable?

3. Consider the following type and function declarations

```
type form =
  Var of string
| And of form list

let rec free_var (f : form) =
  match f with
    Var s -> [s]
  | And fs -> free_vars fs

and free_vars (fs: form list) =
  match fs with
    [] -> []
  | f :: rest -> free_var f @ free_vars rest
;;
```

Is free_var total?

Is free_vars total?

4.

```
let rec f (x:int) =
  if x > 50 then 1 + f (x-1) else g x

and g (y: int) =
  if y > 0 then 1 else f (x-1)
;;
```

Is f total?

Is g total?

5.

```
let f x = … ;;

let g x =
  if f x then 1 else 0
;;
```

What do we need to know about f to know that g is total?

Part II

Give justifications for each of the following equations using the equational rules given in the online notes. Whenever you need to use reflexivity, transitivity, symmetry, congruence, etc., say so.

let inc x = x + 1;;

(1) inc 3 == 4 _____

(2) inc 4 == 5 _____

(3) inc (inc 3) == 5 _____

(4) fun x -> x + 1 == inc _____

(5) for all values v, v + 1 == inc v _____

(6) for all valuable expressions e, e + 1 == inc e _____

Part III

Consider the following code:

```
let multo (x:int option) (y:int option) =
  match (x,y) with
    (Some m, Some n) -> Some ((m + m)*n)
  | (_, _) -> None
;;
```

Prove the following equation holds, step by step, for all `o : int option`

```
multo o o == (match o with Some i -> Some (2*(i*i)) | None -> None)
```

Proof: (*Note: You can start top down, or you can start bottom up, or go from both ends to the middle*)

```
   multo o o
```

==

== (match o with Some i -> Some (2*(i*i)) | None -> None)

Part IV

Consider the following function.

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

Prove using equational reasoning that for all n : int

```
   compose (fun x -> x * 2) (fun y -> y * 8) n
== compose (fun z -> z * 4) (fun w -> w * 4) n
```

Proof (put one reasoning step on each line with a justification):

(*again, recall you can start from the left-hand side and prove to the right; or start on the right-hand side and prove to the left or go from both sides and try to meet in the middle*)

Part V

Consider the functions double and half:

```
let rec double (xs: int list) : int list =
  match xs with
    [] -> []
  | hd :: rest -> hd::hd::double rest
;;

let rec half (xs: int list) : int list =
  match xs with
    [] -> []
  | [x] -> []
  | x::y::rest -> y::half rest
;;
```

(a) Disprove this conjecture: for all l, double(half l) == l.
(Rhetorical question: How does one disprove such a conjecture?)

(b) Prove that for all integer lists l. half (double l) == l.
Proof: By induction on the structure of the list l:

case l = [] To show:
Proof:

case l = hd::tail To show:

IH:

Proof:

(continue on back if necessary)

Part VI

Recall the familiar map and function composition operators:

```
let rec map (f: 'a -> 'b) (xs : 'a list) : 'b list =
  match xs with
      [] -> []
    | hd::tail -> f hd :: map f tail
;;

(* this is the same as the function "compose" defined earlier,
 * but we are using the infix operator % instead this time for fun *)
let (%) (g:'b -> 'c)  (f:'a -> 'b) (x:'a) : 'c =
  g (f x)
;;
```

A common program optimization is to take a series of several map operations and compress them in to a single map operation. Instead of traversing a list multiple times (once for each application of the map operation), one only traverses the list once. More specifically, the following property of map is true (for any total functions f and g with the correct types):

```
map (g % f) == (map g) % (map f)
```

We are going to prove this fact with the aid of the following lemma:

Lemma 1: For all types 'a, 'b and values `f` : 'a -> 'b, if `f` is total then `map f` is valuable and total.

Now the theorem. We will provide each step in the proof of the theorem. It is up to you to provide the correct justification. However, you can omit mentioning reflexivity, transitivity, symmetry or congruence explicitly (just use these laws wherever you see fit from now on without mentioning them).

Theorem 2: For all types 'a, 'b, 'c and for all values `f` : 'a -> 'b and `g` : 'b -> 'c and l : 'a list, if f and g are total then

```
((map g) % (map f)) l == map (g % f) l
```

Proof:
First we write down our assumptions and give them numbers to refer to them:

(1) f is total
(2) g is total

(skip to the next page)

Now, let's write down some initial facts that may be of use to us. Give justifications on the blank lines next to each statement:

(3) map f is valuable and total _____
(4) map g is valuable and total _____
(5) g % f is valuable _____
(6) g % f is total _____
(7) map (g % f) is valuable and total _____

Next, observe the following:

(8) ((map g) % (map f)) l
(9) == (fun x -> map g (map f y)) l _____
(10) == map g (map f l) _____

Therefore, (by transitivity of equality), we only have to prove that:

```
map g (map f l) == map (g % f) l
```

We do the above by induction on the structure of the list l using the standard list template:

Case for l = []:

Must prove in this case: _____
(11) map g (map f []) _____
(12) == (map g []) _____
(13) == [] _____
(14) (map (g % f) []) _____

Case for l = x::xs:

Must prove in this case: _____
Induction hypothesis: _____
(15) map g (map f (x::xs)) _____
(16)== map g (f x :: map f xs) _____
(17)== g (f x) :: map g (map f xs) several comments:_____
_____
(18)== (g % f) x :: map g (map f xs) _____
(19)== (g % f) x :: map (g % f) xs _____
(20)== map (g % f) (x::xs) _____

QED!