Homework 4

Homework Due: Monday, October 14 at 10 am.

The goal for Homework 4 is to use induction to prove the correctness of OCaml programs. OCaml is full of pattern-matching and recursion, and *induction* is the perfect mathematical tool to handle both.

1 Lab Steps to Complete

As usual, you are encouraged to help each other to finish the lab part.

Prepare Your HW4 Directory

```
cd ~/csci2041
git clone git@github.umn.edu:umn-csci-2041-f19/hw4.git
cd hw4
./setup
```

1. **Run `./setup` every time you clone it.** Please follow the instructions to fix any error or warning.
2. **Do NOT edit files on GitHub directly.** I need to call the IT if you accidentally reveal your homework.

Recap: Induction Order

An induction order can be almost any binary relation as long as there are no infinite descending chains. The jargon is "*well-founded relation*" or "*well-founded order*." For example, the less-than relation $<$ on natural numbers is a good order because you cannot keep getting a smaller number without hitting the zero. The order for induction typically matches how we do recursive calls, and the fact that there are no infinite descending chains means that our program will terminate.

Exercise 1.1

Discuss with your neighbors and determine whether the following relations are good orders:

1. The empty relation on the empty collection.
2. Some reflexive relation on a non-empty collection.
3. The less-than relation on integers.
4. The lexicographic order on strings over the English alphabet.

Exercise 1.2

There are many possible good orders for strings over a finite alphabet. You have determined whether the lexicographic order is good. Two good orders are based on string length and substrings:

1. $R(s_1, s_2)$: the length of s_1 is less than the length of s_2.
2. $R(s_1, s_2)$: s_1 is a (strict) substring of s_2.

Can you come up with yet another good order for strings? See if your neighbors have a different answer.

Exercise 1.3

```
let rec length : 'a . 'a list -> int
  = fun l ->
    match l with
    | [] -> 0
    | (x::l') -> 1 + length l'

let rec append : 'a . 'a list -> 'a list -> 'a list
  = fun l1 -> fun l2 ->
    match l1 with
    | [] -> l2
    | x::l1' -> x :: append l1' l2
```

Help each other to finish this proof. You may assume (1) there are no integer overflows and (2) there are no effects. In particular, you may change the evaluation order and assume every expression terminates (evaluating to a value). *You should mark the usage of inductive hypothesis clearly.*

Theorem 1 *Fix an OCaml type* `a`*. For any two OCaml values* `l1` *and* `l2` *of type* `a list`*, we have*

`length (append l1 l2)` = `length l1 + length l2`

Proof:

- Domain: all values of type `a list`.
- Property P(`l1`): for any value `l2` of type `a list`, `length (append l1 l2)` = `length l1 + length l2`.
- Induction order: R(`l1`, `l2`): `l1` is a strict suffix of `l2`.
- Partition: {`[]`}, {`x::l` | `x` is a value of type `a` and `l` is a value of type `a list`}.

1. `l1` = `[]`.

   (fill in the rest)

2. `l1` = `x :: l1'`

   (fill in the rest)

□

Exercise 1.4

If you plan to hand-write your solutions, please practice scanning your documents *now,* and reserve enough time before the homework deadline for scanning your work. Claiming that you have done the homework before the deadline but somehow cannot scan or typeset your solution will not grant you a deadline extension. Otherwise, if you plan to use professional typesetting software such as LaTeX, please move on.

Exercise 1.5

In class, Favonia repeated the word "value" more than 100 times. What might be the difficulty if we replace "values" with "expressions" in the previous exercise? Consider the components of induction. The following is the modified statement:

Statement 2 *Fix an OCaml type* `a`. *For any two OCaml* *expressions* `l1` *and* `l2` *of type* `a list`, *we have*

`length (append l1 l2)` = `length l1 + length l2`

This concludes the lab part of Homework 3.

2 Take-Home Portion

You need to typeset or scan your homework as one single file `hw4.pdf`. It is encouraged to use professional typesetting software such as LaTeX and you can reuse the code in `README.tex` for your work. Otherwise, your hand-writing must be clear and legible, and it must be scanned and committed as one PDF `hw4.pdf`.

Remember, every (general) induction has these components:

Domain: the collection of mathematical objects you are talking about.

Property: a property you want to prove for every object in the domain.

Order: an order on the domain which determines the inductive hypothesis available to you.

Partition: a partition of the domain for you to do case analysis.

Exercise 3.1 [2B 30pts]

Finish this proof. You may assume (1) there are no integer overflows, and (2) there are no effects. In particular, you may change the evaluation order and assume every expression terminates (evaluating to a value). *You should mark the usage of inductive hypothesis clearly.*

```
let rec sum : int list -> int
  = fun l ->
    match l with
    | [] -> 0
    | (x::l') -> x + sum l'
```

Theorem 3 *For any two OCaml values* `l1` *and* `l2` *of type* `int list`, *we have*

`sum (append l1 l2)` = `sum l1 + sum l2`

Proof:

- Domain: all values of type `a list`.
- Property P(`l1`): for any value `l2` of type `int list`, `sum (append l1 l2)` = `sum l1 + sum l2`.
- Induction order: R(`l1`, `l2`): `l1` is a strict suffix of `l2`.
- Partition: {`[]`}, {`x::l` | `x` is a value of type `a` and `l` is a value of type `a list`}.

(fill in the rest)

□

Hints: You can discuss Exercise 1.3 with anyone until you fully understand it, and then do this exercise.

Exercise 3.2 [2B 30pts]

Finish this proof. Again, you may assume (1) there are no integer overflows, and (2) there are no effects. In particular, you may change the evaluation order and assume every expression terminates (evaluating to a value). *You should mark the usage of inductive hypothesis clearly.*

```
let rec snoc : 'a . 'a list -> 'a -> 'a list
  = fun l -> fun x ->
    match l with
    | [] -> [x]
    | (y::l') -> y :: snoc l' x

let rec map : 'a 'b . ('a -> 'b) -> 'a list -> 'b list
  = fun f -> fun l ->
    match l with
    | [] -> []
    | (x::l') -> f x :: map f l'
```

Theorem 4 *Fix two OCaml types* `a` *and* `b`*. For any OCaml value* `f` *of type* `a -> b`*, any OCaml value* `l` *of type* `a list`*, and any OCaml value* `x` *of type* `a`*, we have*

`map f (snoc l x)` = `snoc (map f l) (f x)`

Note: Do not give two variables the same name. It is strictly less helpful.

Exercise 3.3

The goal of this exercise is to prove `reverse` and `reverse2` implement the same function. There are many ways to prove it, and the following is one possibility. Once again, you may assume (1) there are no integer overflows, and (2) there are no effects. In particular, you may change the evaluation order and assume every expression terminates (evaluating to a value). *You should mark the usage of inductive hypothesis clearly.*

Exercise 3.3.1 [2A 20pts, bonus 2B 6pts]

```
let rec reverse : 'a . 'a list -> 'a list
  = fun l ->
    match l with
    | [] -> []
    | (x :: l) -> snoc (reverse l) x

let rec rev_append : 'a . 'a list -> 'a list -> 'a list
  = fun l1 -> fun l2 ->
    match l1 with
    | [] -> l2
    | x::l1' -> rev_append l1' (x :: l2)
```

Prove this theorem:

Theorem 5 *Fix an OCaml type* `a`*. For any two OCaml values* `l1` *and* `l2` *of type* `a list`*, we have*

`rev_append l1 l2` = `append (reverse l1) l2`

Hints: Can you find another OCaml expression equivalent to `append (snoc l3 x) l4` without using `snoc`? Can you use induction to prove the equivalence? Study the proof of the theorem `reverse (reverse l)` = `l`.

Exercise 3.3.2 [2A 10pts, bonus 2B 3pts]

```
let reverse2 : 'a . 'a list -> 'a list
  = fun l -> rev_append l []
```

Prove this theorem:

Theorem 6 *Fix an OCaml type* `a`*. For any OCaml value* `l` *of type* `a list`*, we have*

`reverse l` = `reverse2 l`

Hints: You might want a lemma about `append`.

Anonymous Feedback

We are collecting anonymous feedback throughout the semester.

Due

Homework Due: Monday, October 14 at 10 am.