Great !!

3.1

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::l1 | X is value of type a and l is a value of type a list}

IH: Sum(append l1 l2) = Sum l1 + Sum l2

Case 1: l1 = []

Sum(append [] l2) = Sum(l2) = 0 + Sum(l2) = Sum([]) + Sum(l2)

Case 2: l1 = X :: l1' in Sum(append l1 l2) =>

Sum(append (X::l1') l2) = Sum(X :: append (l1'. l2))

= X + Sum(append (l1' l2)) BY the IH ->

= X + Sum l1' + Sum l2

*the return statement for the call Sum(X::l1') = X + Sum l1'

= Sum (X :: l1') + Sum l2

3.2

Domain: all values of type a list

Property: For any value X of type int list, map f (snoc l x) = snoc (map f l)(f x)

Induction Order: R(l, X) l1 is a strict suffix of X

Partition: {[]}, {h::l' | x is a value of type a and l' is a value of type a list

IH: map f (snoc l x) = snoc (map f l) (f x)

Case 1: l = []

map f (snoc [] x) = map f ([x]) = f x = snoc ([]) (f x)

= snoc (map f []) (f x)

Case 2: l = {h :: l'}

map f (snoc {h::l'} x) = map f (h :: snoc l' x)

= f h :: map f (snoc l' x) BY the IH ->

= f h :: snoc (map f l') (f x)

= snoc (f h :: map f t) (f x)

= snoc (map f {h::t}) (f x)

3.3.1

Thm 5: Fix an OCaml type a. For any 2 OCaml values l1 and l2 of type a list
rev_append l1 l2 = append (reverse l1) l2

Domain: All values of type a list

Property P(l1): For any value l2 of type a list, rev_append l1 l2 = append (reverse l1) l2

Order R(l1, l2): l1 is a strict suffix of l2

Partition: {[]}, {x :: l | x is a value of type a, l is a value of type a list}

Case 1: l1 = []

Prove: rev-append [] l2 = append (reverse []) l2

rev_append [] l2 = l2 = append ([] l2) = append (reverse []) l2

Case 2: l1 = {h :: t}

Prove: rev_append {h :: t} l2 = append (reverse h :: t) l2

= rev_append t {h :: l2}

= append (reverse t) h :: l2 by Inductive Hypothesis (thm 5)

= append (snoc (reverse t) h) l2 by Lemma 1

= append (reverse h :: t) l2

Lemma 1

Domain: all values c and d of type a list

Property P(c): append (snoc (reverse c) h) d = append (reverse c) h :: d

Order R(c, d): c is a strict suffix of d

Partition: {[]}, {h :: c | h is a value of type a and c is a value of type a list}

Case 1: c = []

append (snoc (reverse [])) h) d = append (reverse []) h :: d

= append (snoc [] h) d

= append [h] d

= h :: d by Inductive Hypothesis

= append [] h :: d

= append (reverse []) h :: d

3.3.2

Thm 6: Fix an OCaml type a for any OCaml value l of type a list, we have reverse l = reverse2 l

Domain: all values of type a list

Property P(l): for any value l of type a list, reverse l = reverse2 l

Order R(l1, l2): l1 is a strict suffix of l2

Partition: {[]}, {h::l | h is a value of type a, l is a value of type a list}

Case 1: l = []

reverse [] = [] = reverse2 []

Case 2: l = {h::t}

prove: reverse {h::t} = reverse 2 {h::t}

↳ = append (reverse h::t) [] → by lemma 2

= rev_append h::t [] → by thm5

= reverse 2 h::t

Lemma 2: Fix an OCaml type a for any OCaml value l of type a list and any value h of type a. for append (reverse l) [] = reverse l

Domain: all values l of type a list

Property P(l): append (reverse l) [] = reverse l

Order R(l1, l2): l1 is a strict suffix of l2

Partition: {[]}, {h::l | h is a value of type a, l is a value of type a list}

Case 1: l = []

prove: append (reverse []) [] = reverse []

↳ = append [] [] = [] = reverse []

Case 2: l = {h::t}

prove: append (reverse h::t) [] = reverse h::t

↳ = append (snoc (reverse t) h) []

= snoc (reverse t) h::[]

= snoc (reverse t) h

= reverse h::t