Proofs

Jonas Hübotter

February 13, 2022

1 Exercises

1.1 Structural induction

1. [sheet 15] Prove that

```
map f (concat xss) = concat (map (map f) xss)
```

where

```
map f [] = []
map f (x:xs) = f x : map f xs

concat [] = []
concat (xs:xss) = xs ++ concat xss
```

You may use the lemma `map_append`:

`map f (xs ++ ys) = map f xs ++ map f ys`

2. [endterm 2020] Given the type of natural numbers

```
data Nat = Z | Suc Nat
```

and the following definition of addition on these numbers

```
add Z m = m
add (Suc n) m = Suc (add n m)
```

show that addition is associative by proving the following equation using structural induction:

```
add (add x y) z = add x (add y z)
```

1.2 Case analysis

1. [sheet 13] In this exercise, we consider the datatype `AExp` which models addition and multiplication on integers:

```
data AExp = Val Integer | Add AExp AExp | Mul AExp AExp
  deriving Eq
```

We define a function `eval` to evaluate an expression to an integer, and a function `simp` that simplifies expressions of the form `0 + e` to `e`:

```
eval (Val i) = i
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)

simp (Val i) = Val i
simp (Mul a b) = Mul (simp a) (simp b)
simp (Add a b) = if a == Val 0 then simp b else Add (simp a) (simp b)
```

Your task is to prove that this simplification preserves the value of an expression, i.e. that the following equation holds:

```
eval (simp e) = eval e
```

You may use these familiar axioms, no further rules for arithmetic should be required:

```
axiom addZero: x + 0 = x
axiom zeroAdd: 0 + x = x
```

1.3 Generalization

1. Given the following definitions:

```
data List a = [] | a : List a

reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

itrev [] xs = xs
itrev (x:xs) ys = itrev xs (x:ys)

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Prove the following statement using structural induction:

```
itrev xs [] = reverse xs
```

You may use the following lemmas about `++` in the proof:

```
Lemma app_assoc: (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
Lemma app_empty: xs ++ [] = xs
```

1.4 Extensionality

1. [sheet 7] This exercise is all about the two different fold functions `foldl` and `foldr`, which are defined as follows:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs

foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

The function signatures are similar; however, there is a key difference in their functionality: As the names suggest, `foldl` performs a left-associative and `foldr` a right-associative fold, respectively. More concretely, we have that

```
foldl f z [x1, x2, ..., xn] = (...((z `f` x1) `f` x2) `f` ...) `f` xn

foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ... (xn `f` z)...)
```

Let `f` be a binary operator that is commutative with respect to `a`, i.e. `f x a = f a x` for all `x`, and associative. Prove the statement: `Lemma: foldl f a .=. foldr f a`.

1.5 Computation induction

1. [sheet 6] We define the functions `sum :: Num a => [a] -> a` and `sum2 :: Num a => [a] -> [a] -> a`:

```
sum [] = 0
sum (x:xs) = x + sum xs
sum2 [] [] = 0
sum2 [] (y:ys) = y + sum2 ys []
sum2 (x:xs) ys = x + sum2 xs ys
```

Use computation induction to show that `sum2 xs ys = sum xs + sum ys`.

2 Homework

2.1 Structural induction

1. [sheet 6] We define functions `sum :: Num a => [a] -> a` and `(++) :: [a] -> [a] -> [a]` as follows:

```
sum xs = sum_aux xs 0
sum_aux [] acc = acc
sum_aux (x:xs) acc = sum_aux xs (acc + x)

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Use structural induction to show that

```
sum (xs ++ ys) = sum xs + sum ys
```

2. [sheet 5] We define `snoc :: [a] -> a -> [a]` and `reverse :: [a] -> [a]` as follows:

```
snoc [] y = [y]
snoc (x:xs) y = x : snoc xs y

reverse [] = []
reverse (x:xs) = snoc ( reverse xs ) x
```

   (a) Use structural induction to prove the following equation

   ```
   reverse (snox xs x) = x : reverse xs
   ```

   (b) Use structural induction to prove the following equation

   ```
   reverse (reverse xs) = xs
   ```

3. [sheet 9] Show that the `sumTree` function from problem set 2 indeed works as expected, i.e. prove the equivalence

```
sum (inorder t) = sumTree t
```

using structural induction on trees.

2.2 Generalization

1. [endterm 2020] You are given the following definitions:

```
data Tree a = L | N (Tree a) a (Tree a)

flat L = []
flat (N l x r) = flat l ++ (x : flat r)

app L xs = xs
app (N l x r) xs = app l (x : app r xs)

[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Prove the following statement using structural induction:

```
app t [] = flat t
```

You may use the following lemmas about `++` in the proof:

```
axiom app_assoc: (xs ++ ys) ++ zs .=. xs ++ (ys ++ zs)
axiom app_nil: xs ++ [] .=. xs
axiom nil_app: [] ++ xs .=. xs
```

2.3 Extensionality

1. [sheet 7] Prove the proposition

```
filter p . filter p = filter p
```

where `filter :: (a -> Bool) -> [a] -> [a]` and `(.) :: (b -> c) -> (a -> b) -> (a -> c)` are defined as

```
filter f [] = []
filter f (x:xs) = if f x then x : filter f xs else filter f xs

(f . g) x = f (g x)
```

You may also use the following axioms about if-expressions:

```
axiom if_True: (if True then x else y) .=. x
axiom if_False: (if False then x else y) .=. y
```

2.4 Computation induction

1. Given the following definition of `drop2`:

```
drop2 [] = []
drop2 [x] = [x]
drop2 (x:y:xs) = x : drop2 xs

length [] = 0
length (x:xs) = 1 + length xs
```

And the axioms:

```
axiom: addZeroOne: 0 + 1 .=. 1
axiom: addOneZero: 1 + 0 .=. 1
axiom: addOneOne: 1 + 1 .=. 2
axiom: addAssoc: (a + b) + c .=. b + (b + c)
axiom: divOneTwo: 1 `div` 2 .=. 0
axiom: divTwoTwo: 2 `div` 2 .=. 1
axiom: divMulOneTwo: 1 + (x `div` 2) .=. (x + 2) `div` 2
```

Prove the following statement using `drop2`-induction:

```
length (drop2 xs) = (length xs + 1) `div` 2
```

2. [sheet 6] We define:

```
length [] = 0
length (x:xs) = 1 + length xs
```

```
countGt [] ys = 0
countGt (x:xs) [] = length (x:xs)
countGt (x:xs) (y:ys) = if x > y then 1 + countGt (x:xs) ys
  else countGt (y:ys) xs
```

Show that `countGt xs ys <= length xs + length ys` using computation induction.

Note: Given a rule `P x ==> y <= z` with name `myRule` and a proof `p` of `P x`, you can use `(by myRule OF p` to apply the inequality between `y` and `z`. For example:

```
axiom leAddMono : y <= z ==> x + y <= x + z
axiom zeroLeOne : 0 <= 1
Lemma : 0 + 0 <= 0 + 1
Proof
                                 0 + 0
  (by leAddMono OF zeroLeOne) <= 0 + 1
QED
```