8 Reasoning and calculating

The exercises all deal with inductive and equational reasoning. There should be plenty of material to practice on. All it takes is being able to apply rewrite steps; the difficulty is picking the right ones. Use a structured approach, re-use the template from Exercise 8.1, and see Hint 1.

Rewrite rules that you are asked to prove in one exercise **may be used in subsequent exercises** by simply referring to the exercise that introduced them. You don't have to prove those every time you use them. The same holds for definitions introduced in earlier exercises.

Exercise 8.1 (*Warm-up:* Understanding proofs, `ProofTemplate.lhs`).
In the lecture, it was proven that *finite lists* form a so-called '*monoid*' with operation `++` and identity element `[]`. That is, for all *finite* lists `xs`, `ys`, `zs`, the following three properties were proven:

```
left-identity:  [] ++ xs = xs
right-identity: xs ++ [] = xs
associativity:  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

1. Which of these properties follows from the definition of `++`, and which require induction?
2. Practice your proving skills by re-proving the two properties that require induction; don't just copy the steps, try to *reproduce them* yourself.

Friendly note: this exercise is very similar to many that follow; but here you have the benefit that you can check your answer. Copy the proof structure given in the template `ProofTemplate.lhs` for your subsequent proofs. Proving is like programming: you *learn by doing*.

Exercise 8.2 (*Warm-up:* Proofs and synthesis, `ReverseCat.lhs`).
During the lecture, the function `reverseCat` was *derived* using program synthesis. `reverseCat` is a helper function used in a tail-recursive implementation of `reverse` (cf. Exercise 3.2):

```
reverse :: [a] → [a]
reverse []          = []
reverse (x:xs)      = reverse xs ++ [x]

reverse' :: [a] → [a]
reverse' xs         = reverseCat xs []

reverseCat :: [a] → [a] → [a]
reverseCat []     ys = ys
reverseCat (x:xs) ys = reverseCat xs (x:ys)
```

1. Prove that `reverseCat xs ys = reverse xs ++ ys`, using induction.
2. Prove that `reverse xs = reverse' xs`.

(The proofs should feel similar to the *program synthesis* steps for `reverseCat`!)

Exercise 8.3 (*Warm-up:* induction on lists, `ListInduction.lhs`).
Consider the three following function definitions:

```
(.) :: (b → c) → (a → b) → a → c
(f . g) x = f (g x)                 (0)

(++) :: [a] → [a] → [a]
[]     ++  ys = ys                  (1)
(x:xs) ++  ys = x : (xs ++ ys)      (2)

map :: (a → b) → [a] → [b]
map f []      = []                  (3)
map f (x:xs)  = f x : map f xs      (4)

concat :: [[a]] → [a]
concat []     = []                  (5)
concat (x:xs) = x ++ concat xs      (6)
```

1. Prove the following proposition for all (finite) lists xs, and functions f and g:

   `map (f . g) xs = map f (map g xs)`

2. Prove the following proposition for all (finite) lists as and bs, and functions f (carefully consider on what list you apply the induction!).

   `map f (as ++ bs) = (map f as) ++ (map f bs)`

3. Which type must `xs` have for the following proposition to hold? Prove that it does for (finite) lists `xs` using structural induction.

   `concat (map (map f) xs) = map f (concat xs)`

Exercise 8.4 (*Warm-up:* Proofs involving folds, `FoldrInduction.lhs`).
Given the following definitions:

```
foldr :: (a → b → b) → b → [a] → b
foldr f b []     = b                (7)
foldr f b (x:xs) = f x (foldr f b xs)  (8)

compose :: [a → a] → a → a
compose [] = id                     (9)
compose (f:fs) = f . compose fs     (10)
```

In Exercise 5.4, we essentially hypothesized that for all functions `f`, initial elements `b`, and *finite lists* `xs`, it is the case that `foldr f b xs = compose (map f xs) b`. Use induction to prove it!

Exercise 8.5 (*Warm-up*: Induction over algebraic data types, `TreeInduction.lhs`).
Every recursive datatype comes with a pattern of induction. This pattern is very similar to the *recursive design pattern* for that type. For example, for lists, a proof has two cases: the empty list `[]`, and the *cons*-case `(x:xs)`. Since in the latter case we (recursively) encounter another list `xs`, we may assume that whatever we are trying to prove already holds for that list (but only that list)—which is the *induction hypothesis*.

If we had defined lists ourselves, the situation would be the same:

```
data List a = Nil | Cons a (List a)
```

Again there are two cases to prove a property `P` for all objects of type `List a`:

Base case Show that the property holds for `Nil`:

```
P(Nil)
```

Inductive step assuming the property holds for some `lst`, show that it holds for the `Cons` of `lst`:

`P(lst)` $\Longrightarrow$ for all `x`, `P(Cons x lst)`

Here `P(lst)` is the *induction hypothesis*.

1. Similarly explain the induction scheme for binary trees:

   ```
   data Tree a = Leaf | Node a (Tree a) (Tree a)
   ```

   (Hint: you get an *induction hypothesis* for two objects instead of one.)

2. Given these definitions:

   ```
   leaves :: Tree a → Int
   leaves Leaf = 1
   leaves (Node _ l r) = leaves l + leaves r

   nodes :: Tree a → Int
   nodes Leaf = 0
   nodes (Node _ l r) = 1 + nodes l + nodes r
   ```

   Prove by induction that for all *finite trees* `t`, `leaves t = nodes t + 1`.

Exercise 8.6 (*Mandatory:* Induction over algebraic data types trees, `Btree.lhs`).
In this exercise we use the following type and function definitions:

```
data Btree a = Tip a | Bin (Btree a) (Btree a)

mapBtree :: (a → b) → Btree a → Btree b
mapBtree f (Tip a)     = Tip (f a)
mapBtree f (Bin t1 t2) = Bin (mapBtree f t1) (mapBtree f t2)

tips :: (Btree a) → [a]
tips (Tip x) = [x]
tips (Bin as bs) = tips as ++ tips bs
```

Prove the following proposition for any function f and any (finite) Btree t:

```
map f (tips t) = tips (mapBtree f t)
```

(Like in the previous exercise, use an appropriate induction pattern.)

Exercise 8.7 (*Mandatory*: Program derivation, `ProgramDerivation.hs`). The implementation of `inorder` from Exercise 4.5:

```
inorder :: Tree a → [a]
inorder Leaf            =  []
inorder (Node x lt rt) =  inorder lt ++ [x] ++ inorder rt
```

has a quadratic running time because of the repeated invocations of list concatenation—recall that the running time of `++` is linear in the length of its first argument. To improve the running time we solve a more complicated task (see also Exercise 8.2).

```
inorderCat t xs = inorder t ++ xs
```

At first sight, you will wonder why we are making our problem harder. This technique is known as *inventor's paradox*, or *strengthening the induction hypothesis.* The important observation is that while the problem is more difficult, the recursive call gives an induction hypothesis that is also much stronger, which can make proving the inductive step *easier.*

1. Derive an implementation of `inorderCat` from the specification above using equational reasoning; and then define `inorder` in terms of `inorderCat`.

2. Test the resulting program on a few test cases. (Tip: use the function

   ```
   skewed :: Integer → Tree ()
   ```

   that generates a left-skewed tree of the given height.) Is the new implementation of `inorder` actually more efficient?

3. Repeat the above procedure to create a more efficient version of `elems`:

   ```
   elems :: Tree a → [a]
   elems Leaf           = []
   elems (Node x lt rt) = x : elems lt ++ elems rt
   ```

Exercise 8.8 (*Mandatory:* `foldr` fusion, `FoldrFusion.lhs`).
Like `foldr` captures a common recursion scheme (canned recursion), `foldr` fusion captures a common induction scheme (canned induction). The `foldr` fusion law for states that if

```
f (g x y) = h x (f y)
```

for all `x` and all `y`, then

```
f . foldr g z = foldr h (f z)
```

Using `foldr` fusion, we can prove another, easier to use (but less general) `foldr`-`map` fusion law:

```
foldr p e . map q  =  foldr (p . q) e
```

To do this, we will first need to write `map` in terms of `foldr`.

1. Give a definition of `map` in terms of `foldr`, by completing the following definition:

   ```
   map f = foldr (step f) _todo_1_
     where step f x xs = _todo_2_
   ```

   In the rest of the exercise you may assume that `map` is defined in this way.

2. Now we can use `foldr` fusion, to prove that, for all `p`, `q`, and `e`:

   ```
   foldr p e . map q  =  foldr (p . q) e
   ```

   What expressions should you choose for `f`, `g`, `h`, and `z` to make this statement match the `foldr` fusion law?

3. Complete the proof of the statement in `FoldrFusion.lhs`.

   Note: When replacing a metavariable with a *compound expression* (e.g. here `h =⇒ a . b`), you must *parenthesize* the substituted parameter in the resulting term (e.g. here replace `h` with `(a . b)`), else you change its semantics.

4. Using `foldr` fusion above is only allowed if we can prove `f (g x y) = h x (f y)` for all `x` and `y`. Write down this condition using the values for `f`, `g`, `h` you chose above.

5. Prove that the use of `foldr` fusion was allowed, thus finishing the proof. You do not need induction, but you will need some definitions from Exercise 8.3 and 8.4.

Now we can *apply* the `foldr`-`map` fusion law we just proved. The second part of this exercise can be completed independently from the first part.

6. Use `foldr`-`map` fusion to prove:

   ```
   map (f . g) = map f . map g
   ```

   (If you feel adventurous, you can also establish this using `foldr` fusion directly.)

   Note: you maybe first need to prove that `step (f . g) = step f . g`

7. *Optional*: Use `foldr` fusion and `foldr`-`map` fusion to prove the 'bookkeeping law':

   ```
   mconcat . concat = mconcat . map mconcat
   ```

You may use the *monoid laws*; see Exercise 8.9. From that exercise you may also use:

```
mconcat (x ++ y) = mconcat x <> mconcat y
```

Exercise 8.9 (*Extra:* Proofs involving monoids, `MonoidInduction.lhs`).
The term *monoid* is just a fancy mathematical name to describe a data type coupled with an operation. For something to be a monoid, this operation operation has to be *associative* and it has to have an *identity element*. Examples are `Integer` coupled with the `+` operation (identity element 0), `Integer` with `*` (identity element 1), lists with `++` (Exercise 8.1), and functions of type `a → a` with function composition.

In Haskell there exists a `Monoid` type class so we can write polymorphic functions that work for all monoids. Formally, suppose that for some data type we have an operator `<>`, and an identity element `mempty`, for it to be a monoid, the following *monoid laws* have to hold:

```
left-identity:  mempty <> x = x
right-identity: x <> mempty = x
associativity:  x <> (y <> z) = (x <> y) <> z
```

Of course, given a list of objects of a monoid, we can repeatedly apply the `<>` operator to collapse it; this is called *monoid concatenation*, defined as:

```
mconcat :: (Monoid a) ⇒ [a] → a
mconcat []     = mempty
mconcat (x:xs) = x <> mconcat xs
```

or equivalently:

```
mconcat :: (Monoid a) ⇒ [a] → a
mconcat = foldr (<>) mempty
```

The evaluation order (i.e. whether we use `foldr` or `foldl`) for monoids actually *doesn't matter*. We will prove that! In the below exercises, you may assume that `(<>)` and `mempty` satisfy the *monoid laws*.

1. Prove using induction that if `xs` and `ys` are *finite lists* of a type `a` that is a `Monoid` instance:

   ```
   mconcat (xs ++ ys) = mconcat xs <> mconcat ys
   ```

   Again, as in Exercise 8.3, carefully consider the list you are going to run the induction on.

2. Using this definition of `foldl`:

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

   Prove that for all elements `x`, `y` and *finite lists* `xs`:

   ```
   foldl (<>) (x<>y) xs = x <> foldl (<>) y xs
   ```

3. Use the previous result to prove that `foldr` and `foldl` are functionally equivalent as a reducing operation, by proving that for all *finite lists* `xs`.

```
foldl (<>) mempty xs = foldr (<>) mempty xs
```

Of course, as seen in Exercise 5.9 and Exercise 6.4, there can still be dramatic differences in efficiency, which might lead you to prefer one over the other.

Exercise 8.10 (*Extra:* Making proofs in Agda, `ReverseCat.agda`). Instead of doing proofs about functional programs by hand, wouldn't it be nice if the computer could help use? Indeed, we can use a proof assistant to check our proofs for us and to help with writing them.

In this assignment we will use Agda, which is a language very similar to Haskell. In Agda, equality statement can be expressed as types, and a proof of such a statement is a value of this equality type. For example, `prop-id : ∀x →id x ≡x` is a proof that for all x, `id x` is equal to `x`. And the proof is named `prop-id`.

First you will need to install Agda, by following the instructions on https://agda.readthedocs.io/en/latest/getting-started/installation.html.

1. Open ReverseCat.agda in emacs:

   ```
   emacs ReverseCat.agda
   ```

   For those of you unfamiliar with this editor, here is a quick cheat-sheet on using it:

Ctrl+G	Cancel
Ctrl+X, Ctrl+S	Save file
Ctrl+X, Ctrl+C	Quit emacs
Ctrl+C, Ctrl+L	Load file into agda proof checker
Ctrl+C, Ctrl+F	Go to next hole
Ctrl+C, Ctrl+B	Go to previous hole
Ctrl+C, [comma]	Show hole under cursor
Ctrl+C, Ctrl+C	Make a case distinction
Ctrl+C, Ctrl+A	Automatically fill in a hole
Ctrl+C, [space]	Fill in hole with what you wrote

2. Look through the file, and see if you understand what is going on. You can see in this file that Agda looks very similar to Haskell, with some minor syntactic differences:

Haskell	Agda
`value :: Type`	`value : Type`
`head : tail`	`head :: tail`
`list :: [a]`	`list : List A`
`list = [x]`	`list = [ x ] --spaces around all operators`
`(++)`	`_++_`

3. The file contains a proof that `[]` is the right identity of `++`. And this proposition is called `++-right-identity`. Compare this to the proof of the same proposition in `ProofTemplate.lhs`. Hopefully you agree that the two proofs look very similar.

Here is a conversion chart to help you out:

Paper proof	Agda
Start of proof	`begin`
End of proof	∎
Case `x = []`: ..	`++-right-identity [] = ..--pattern matching`
`x = y --reason`	`x ≡⟨ reason ⟩ y`
`I.H.`	recursive call to `++-right-identity`
definition of `(++)`	`++-def-1`
definition in other direction	`sym ++-def-1`
`.. x ..= ..y ..`	`cong! proof --if proof : x ≡y`

4. Now, adapt the proof from Exercise 8.2.1 to agda. That is, give a definition for

   ```
   prop-reverseCat-reverse : ∀ xs ys → reverseCat xs ys  ≡  reverse xs ++ ys
   ```

   Note that the proofs uses some uncommon Unicode symbols. You can type them using backslash: `\::` gives `∷` , `\==` gives `≡` , `\<  \>` gives `⟨ ⟩` , `\qed` gives ∎,

5. Adapt the proof from Exercise 8.2.2 to agda. That is, give a definition for

   ```
   prop-reverse'-reverse : ∀ xs → reverse' xs  ≡  reverse xs
   ```

Exercise 8.11 (*Extra:* More induction proofs, loose ends).
In Exercise 8.8, we introduced the `foldr` fusion law and `foldr-map` fusion law.

1. Show that the `foldr` fusion law holds for all *finite* lists. That is, given some functions `f`, `g` such that for all `x`, `y`:

   ```
   f (g x y) = h x (f y)
   ```

   Prove **using induction** that for all *finite lists* `xs`:

   ```
   f (foldr g e xs) = foldr h (f e) xs
   ```

2. Prove the `foldr-map` fusion law **using induction**; i.e. show that for all *finite lists* `xs`.

   ```
   foldr g e (map f xs) = foldr (g . f) e xs
   ```

   Which style of proof do you prefer?

3. Exercise 8.8 also casually stated that `map` can be written using `foldr`:

   ```
   map f = foldr (\x xs→f x : xs) []
   ```

   Prove it!

Hints to practitioners 1. Equational proofs and derivations shown in textbooks are often the result of a lot of polishing—like the proofs in the lectures. But proving, like programming, is not a spectator sport. So perhaps you appreciate some advice on conducting proofs.

To show the equation `f = g` one typically starts at both ends, and tries to meet in the middle. Write `f` at the top of a text document (or piece of scrap paper), and `g` at the bottom. Apply obvious rewrites such as plugging in definitions going downwards from `f`, or going upwards from `g`.

Perhaps you can identify an intermediate goal where you can apply some rewrite rule you know you need (such as the induction hypothesis!); sometimes you are lucky and this rewrite rule will suggest itself when you get to the 'middle part' of your proof. Whenever you apply a rewrite rule, underline the expression you are re-writing and comment on what rule allows this rewrite—this makes reading the proof easier and also acts as a double-check.

When the proof is finalized, check whether all the rewrite rules are sound, and whether you have used all of the induction hypotheses. If this is not the case, there is some chance that either the proof is wrong, or that you didn't need to use induction at all!