

Programming Languages

Practicals 3. Definition and Proof by Induction

Shin-Cheng Mu

Autumn 2023

1. Prove that *length* distributes into (*++*):

$$\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys .$$

2. Prove: $\text{sum} \cdot \text{concat} = \text{sum} \cdot \text{map sum}$.

3. Prove: $\text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter } (p \cdot f)$.

Hint: for calculation, it might be easier to use this definition of *filter*:

$$\begin{aligned} \text{filter } p [] &= [] \\ \text{filter } p (x : xs) &= \text{if } p\ x \text{ then } x : \text{filter } p\ xs \\ &\quad \text{else } \text{filter } p\ xs \end{aligned}$$

and use the law that in the world of total functions we have:

$$f (\text{if } q \text{ then } e_1 \text{ else } e_2) = \text{if } q \text{ then } f\ e_1 \text{ else } f\ e_2$$

You may also carry out the proof using the definition of *filter* using guards:

$$\begin{aligned} \dots \\ \text{filter } p (x : xs) &\mid p\ x = \dots \\ &\mid \text{otherwise} = \dots \end{aligned}$$

You will then have to distinguish between the two cases: $p\ x$ and $\neg (p\ x)$, which makes the proof more fragmented. Both proofs are okay, however.

4. Reflecting on the law we used in the previous exercise:

$$f (\text{if } q \text{ then } e_1 \text{ else } e_2) = \text{if } q \text{ then } f\ e_1 \text{ else } f\ e_2$$

Can you think of a counterexample to the law above, when we allow the presence of \perp ? What additional constraint shall we impose on f to make the law true?

5. Prove: $\text{take } n\ xs ++ \text{drop } n\ xs = xs$, for all n and xs .

6. Define a function $\text{fan} :: a \rightarrow \text{List } a \rightarrow \text{List } (\text{List } a)$ such that $\text{fan } x \text{ } xs$ inserts x into the 0th, 1st... n th positions of xs , where n is the length of xs . For example:

$$\text{fan } 5 \text{ } [1, 2, 3, 4] = [[5, 1, 2, 3, 4], [1, 5, 2, 3, 4], [1, 2, 5, 3, 4], [1, 2, 3, 5, 4], [1, 2, 3, 4, 5]] \text{ .}$$

7. Prove: $\text{map } (\text{map } f) \cdot \text{fan } x = \text{fan } (f \text{ } x) \cdot \text{map } f$, for all f and x . **Hint:** you will need the *map-fusion* law, and to spot that $\text{map } f \cdot (y :) = (f \text{ } y :) \cdot \text{map } f$ (why?).
8. Define $\text{perms} :: \text{List } a \rightarrow \text{List } (\text{List } a)$ that returns all permutations of the input list. For example:

$$\text{perms } [1, 2, 3] = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]] \text{ .}$$

You will need several auxiliary functions defined in the lectures and in the exercises.

9. Prove: $\text{map } (\text{map } f) \cdot \text{perm} = \text{perm} \cdot \text{map } f$. You may need previously proved results, as well as a property about *concat* and *map*: for all g , we have $\text{map } g \cdot \text{concat} = \text{concat} \cdot \text{map } (\text{map } g)$.
10. Define $\text{inits} :: \text{List } a \rightarrow \text{List } (\text{List } a)$ that returns all prefixes of the input list.

$$\text{inits } \text{"abcde"} = [\text{"", "a", "ab", "abc", "abcd", "abcde"}].$$

Hint: the empty list has *one* prefix: the empty list. The solution has been given in the lecture. Please try it again yourself.

11. Define $\text{tails} :: \text{List } a \rightarrow \text{List } (\text{List } a)$ that returns all suffixes of the input list.

$$\text{tails } \text{"abcde"} = [\text{"abcde", "bcde", "cde", "de", "e", ""}].$$

Hint: the empty list has *one* suffix: the empty list. The solution has been given in the lecture. Please try it again yourself.

12. The function $\text{splits} :: \text{List } a \rightarrow \text{List } (\text{List } a, \text{List } a)$ returns all the ways a list can be split into two. For example,

$$\text{splits } [1, 2, 3, 4] = [([], [1, 2, 3, 4]), ([1], [2, 3, 4]), ([1, 2], [3, 4]), ([1, 2, 3], [4]), ([1, 2, 3, 4], [])] \text{ .}$$

Define *splits* inductively on the input list. **Hint:** you may find it useful to define, in a *where*-clause, an auxiliary function $f \text{ } (ys, zs) = \dots$ that matches pairs. Or you may simply use $(\lambda \text{ } (ys, zs) \rightarrow \dots)$.

13. An *interleaving* of two lists xs and ys is a permutation of the elements of both lists such that the members of xs appear in their original order, and so does the members of ys . Define $\text{interleave} :: \text{List } a \rightarrow \text{List } a \rightarrow \text{List } (\text{List } a)$ such that $\text{interleave } xs \text{ } ys$ is the list of interleavings of xs and ys . For example, $\text{interleave } [1, 2, 3] \text{ } [4, 5]$ yields:

$$[[1, 2, 3, 4, 5], [1, 2, 4, 3, 5], [1, 2, 4, 5, 3], [1, 4, 2, 3, 5], [1, 4, 2, 5, 3], [1, 4, 5, 2, 3], [4, 1, 2, 3, 5], [4, 1, 2, 5, 3], [4, 1, 5, 2, 3], [4, 5, 1, 2, 3]].$$

14. A list ys is a *sublist* of xs if we can obtain ys by removing zero or more elements from xs . For example, $[2, 4]$ is a sublist of $[1, 2, 3, 4]$, while $[3, 2]$ is *not*. The list of all sublists of $[1, 2, 3]$ is:

$[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]$.

Define a function $sublist :: List\ a \rightarrow List\ (List\ a)$ that computes the list of all sublists of the given list. **Hint:** to form a sublist of xs , each element of xs could either be kept or dropped.

15. Consider the following datatype for internally labelled binary trees:

data $Tree\ a = Null \mid Node\ a\ (Tree\ a)\ (Tree\ a)$.

- (a) Given $(\downarrow) :: Nat \rightarrow Nat \rightarrow Nat$, which yields the smaller one of its arguments, define $minT :: Tree\ Nat \rightarrow Nat$, which computes the minimal element in a tree. (Note: (\downarrow) is actually called *min* in the standard library. In the lecture we use the symbol (\downarrow) to be brief.)
- (b) Define $mapT :: (a \rightarrow b) \rightarrow Tree\ a \rightarrow Tree\ b$, which applies the functional argument to each element in a tree.
- (c) Can you define (\downarrow) inductively on Nat ?
- (d) Prove that for all n and t , $minT\ (mapT\ (n+) \ t) = n + minT\ t$. That is, $minT \cdot mapT\ (n+) = (n+) \cdot minT$.