# 7 Reasoning and calculating

**Exercise 7.1** (Warm-up: induction). Recall the implementation of insertion sort from the very first lecture (§0.4).

```
insert :: (Ord a) ⇒ a → [a] → [a]
insert a []    =  [a]
insert a (b : xs)
  | a ≤ b      =  a : b : xs
  | otherwise  =  b : insert a xs

insertionSort :: (Ord a) ⇒ [a] → [a]
insertionSort []        =  []
insertionSort (x : xs)  =  insert x (insertionSort xs)
```

Show the partial correctness of `insert` and `insertionSort`:

```
ordered xs ⟹ ordered (insert x xs)
ordered (insertionSort xs)
```

Why *partial* correctness? Well, we do not capture that the output list is a permutation of the input list. (The definitions `insert x xs = []` and `insertionSort xs = []` trivially satisfy the specification above.) In case you feel adventurous: how would you capture the latter property?

**Exercise 7.2** (Pencil and paper: induction).

1. We have emphasized in the lectures that every datatype comes with a pattern of induction. What is the induction scheme for `Tree elem` i.e. for binary trees?
2. Define functions that count the number of inner nodes (`Node`) and the number of outer nodes (`Empty`). Prove by induction that the latter number is one more than the former. (Put differently, show that a binary tree of size `n` has `n + 1` leaves.)
3. Show by induction

$$2^{\texttt{minHeight t}} - 1 \leq \texttt{size t} \leq 2^{\texttt{maxHeight t}} - 1$$

**Exercise 7.3** (Pencil and paper: induction on lists). Consider the two following function definitions:

```
(++) :: [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 x          (6)
```

1. 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)
```

2. Prove the following proposition using induction on the structure of list *xs* (assuming that *xs* is a finite list):

```
flatten (map (map f) xs) = map f (flatten xs)
```

**Exercise 7.4** (Pencil and paper: lists and trees). 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)                              (1)
mapBTree f (Bin t1 t2) = Bin (mapBTree f t1) (mapBTree f t2)   (2)

foldBTree :: (a → a → a) → BTree a → a
foldBTree f (Tip x)     = x                                     (3)
foldBTree f (Bin t1 t2) = f (foldBTree f t1) (foldBTree f t2)  (4)

tips :: (BTree a) → [a]
tips t = foldBTree (++) (mapBTree (:[]) t)                      (5)
```

Explain what the functions `foldbtree` and `tips` do. Prove the following proposition for any function *f* and any (finite) binary tree *t*:

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

*Hint*: you may need one of the lemmas of the previous exercise.

**Exercise 7.5** (Pencil and paper: program derivation). The implementation of `inorder`

```
inorder :: Tree elem → [elem]
inorder Empty        =  []
inorder (Node l a r)  =  inorder l ++ [a] ++ inorder r
```

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 `reverseCat` and Exercise 7.7 below):

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

At first sight, it may seem slightly paradoxical that a more difficult problem is actually easier i.e. more efficient to solve. This is why this technique is known as *inventor's paradox* or, more prosaically, as *strengthening the induction assumption*. The important observation is that while the problem is more difficult, the recursive call (or the induction assumption) is also more powerful—remember, you only have to program a step (or prove an implication)!

1. Derive an implementation of `inorderCat` from the specification above (7.5) and then define `inorder` in terms of `inorderCat`.

2. Test the resulting program on a few test cases. (You may want to define a 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 exercise for `preorder` and `postorder`.

4. What is the relation of the derivation to an inductive proof? Put differently, can you rearrange the derivation into an inductive proof?

**Exercise 7.6** (Pencil and paper: `foldr` fusion). Like `foldr` captures a common recursion scheme (canned recursion), `foldr` fusion captures a common induction scheme (canned induction). The purpose of this exercise is to train the use of canned induction.

1. Prove the `foldr-map` fusion law:

   $$\texttt{foldr g e} \circ \texttt{map f} = \texttt{foldr (g} \circ \texttt{f) e}$$

   Of course, you should *not* use induction. Instead, apply `foldr` fusion making use of the fact that `map` can be defined in terms of `foldr`.

2. Use `foldr-map` fusion to prove the second functor law:

   $$\texttt{map (f} \circ \texttt{g)} = \texttt{map f} \circ \texttt{map g}$$

   (There is actually a second option: alternatively, you can establish the functor law using `foldr` fusion.)

3. Use `foldr-map` fusion to prove the 'bookkeeping law':

   $$\texttt{reduce} \circ \texttt{concat} = \texttt{reduce} \circ \texttt{map reduce}$$

   Here we assume that `reduce = foldr (<>) mempty`. In what sense does the bookkeeping law generalize the second homomorphism condition? *Hint:* specialize the law to 2-element lists.

**Exercise 7.7** (Worked example: program derivation, `Text.lhs`). Haskell's `show` method converts a value into its string representation. The approach is, however, a tad simplistic and the resulting string is usually not fit for human consumption. To illustrate, consider testing the function `balanced` of Exercise 4.3. Showing `balanced [1 .. 6]` produces:

```
Node (Node Empty 1 (Node Empty 2 Empty)) 3 (Node (Node Empty 4 Empty) 5
                                                 (Node Empty 6 Empty))
```

The purpose of this exercise is to implement a small library for *pretty printing* that allows us to emphasize the *structure* of data. By contrast, pretty printing `balanced [1 .. 6]` produces:

```
Node
    Node
        Empty
        1
        Node
            Empty
            2
            Empty
    3
    Node
        Node
            Empty
            4
            Empty
        5
        Node
            Empty
            6
            Empty
```

The nesting level is nicely indicated through indentation.

The pretty printing interface is minimalist by design: it introduces a datatype of "text with indentation" and a handful of operations.

```
data Text'

infixr 5 <++>
text    :: String → Text       -- without '\n'
nl      :: Text
indent  :: Int → Text → Text
(<++>)  :: Text → Text → Text

render  :: Text → String
```

The function `text` converts a string into a text where the string must not contain any newline characters. Line breaks are introduced explicitly using `nl`. The function call `indent i t` inserts `i` spaces *after* each line break in `t`. The operator `<++>` is the counterpart of `++`: it concatenates two pieces of text. Finally, `render` transforms a text back to a string. The pretty printer for binary trees illustrates the use of the combinators.

```
prettyTree :: (Show elem) ⇒ Tree elem → Text
```

```
prettyTree Empty
  =  text "Empty"
prettyTree (Node l a r)
  =  indent 4 (  text "Node"    <++> nl <++>
                 prettyTree l   <++> nl <++>
                 text (show a)  <++> nl <++>
                 prettyTree r)
```

The string `"Node"`, the left sub-tree, the root element, and the right sub-tree are shown on separate lines using an indentation of `4` spaces.

Let us nail down the semantics of the combinators by providing a reference implementation. The implementation is actually interesting in its own right as it uses the *interpreter design pattern*. Each operation that *produces* text is implemented by a constructor i.e. the operation does (almost) nothing.

```
data Text  =  Text String      -- without '\n'
           |  Nl
           |  Indent Int Text
           |  Text :++ Text
deriving (Show)

text    =  Text
nl      =  Nl
indent  =  Indent
(<++>)    =  (:++)
```

There is only one active operation: `render`, which *consumes* a text. It can be seen as an interpreter, which interprets the other "commands".

```
render (Text s)      =  s
render (Nl)          =  "\n"
render (Indent i d)  =  tab i (render d)
render (d1 :++ d2)   =  render d1 ++ render d2
```

The helper function `tab i s` inserts `i` spaces *after* each newline in `s`.

```
tab :: Int → String → String
tab _i ""        =  ""
tab i (c : s)
    | c == '\n'  =  c : replicate i ' ' ++ tab i s
    | otherwise  =  c :                   tab i s
```

An alternative or, perhaps, complimentary approach to providing a reference implementation is to list algebraic properties of the combinators. For example, we expect that `text ""` and `<++>` form a monoid and that `render` is a monoid homomorphism. Table 2 provides a comprehensive list of properties. Is the list exhaustive? These properties are, in particular, useful for optimizing pretty printers e.g. Equation (??) applied from left to right replaces two indentations by a single one. (As an aside, the reference implementation does *not* satisfy all of these laws. Many of them are only valid *under observation*: instead of `d1 = d2` we only have `render d1 = render d2`.)

1. The reference implementation is quite slow. (Do you see why?) Like in Exercise 7.5, we can improve the performance by solving a more complicated task (applying the *inventor's paradox*): `renderWith doc i x = render (indent i doc) ++ x`

```
      text "" <++> d = d                                  (1)
      d <++> text "" = d                                  (2)
d1 <++> (d2 <++> d3) = (d1 <++> d2) <++> d3                (3)
          indent 0 d = d                                  (4)
   indent i (text s) = text s                             (5)
        indent i nl = nl <++> text (replicate i ' ')      (6)
indent i (indent j d) = indent (i + j) d                   (7)
indent i (d1 <++> d2) = indent i d1 <++> indent i d2       (8)
     render (text s) = s                                   (9)
           render nl = "n"                                 (10)
 render (d1 <++> d2) = render d1 ++ render d2              (11)
```

Table 2: Properties of the pretty printing combinators.

The specified function `renderWith` does several things at a time: it indents a text, transforms the indented text into a string, and then appends another string to the result.

Derive an implementation of `renderWith` from the specification above (1) and then define `render` in terms of `renderWith`. Make use of the properties listed in Table 2. (Do you need all of them?)

2. An advantage of the using the *interpreter design pattern* is that we can easily debug pretty printers: showing `pretty (balanced [1..6])` produces:

```
Indent 4 (Text "Node" :++ (Nl :++ (Indent 4 (Text "Node" :++ (Nl :++
(Text "Empty" :++ (Nl :++ (Text "1" :++ (Nl :++ Indent 4 (Text "Node"
:++ (Nl :++ (Text "Empty" :++ (Nl :++ (Text "2" :++ (Nl :++ Text
"Empty")))))))))) :++ (Nl :++ (Text "3" :++ (Nl :++ Indent 4 (Text
"Node" :++ (Nl :++ (Indent 4 (Text "Node" :++ (Nl :++ (Text "Empty"
:++ (Nl :++ (Text "4" :++ (Nl :++ Text "Empty")))))) :++ (Nl :++ (Text
"5" :++ (Nl :++ Indent 4 (Text "Node" :++ (Nl :++ (Text "Empty" :++
(Nl :++ (Text "6" :++ (Nl :++ Text "Empty"))))))))))))))))
```

OK, perhaps we should pretty print the pretty printing combinators. Implement a pretty printer for the type `Text`. (*For geeks:* to debug the pretty printer for `Text` we can pretty print its output. But the result is again an element of `Text`, so we can apply the pretty printer a second time. It's fun to iterate this process a few more times ...)

3. A slight disadvantage of the using the *interpreter design pattern* is the interpretive overhead. Remove the original definitions of `text`, `nl`, `indent`, and `<>`, which define the operations as constructors. Instead, implement the operations by actual functions. Derive the implementations from the following specifications:

```
text s                          = renderWith (Text s)
nl                              = renderWith (Nl)
indent i (renderWith d)         = renderWith (Indent i d)
renderWith d1 <++> renderWith d2 = renderWith (d1 :++ d2)|
```

1. *Optional:* design a type class `Pretty`, a replacement for `Show`, that builds on the pretty printing library.

**Exercise 7.8** (Programming and program derivation). Here are some problems to work on, just in case you need some inspiration for the seasonal break.

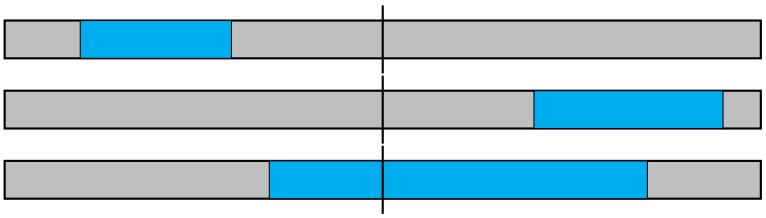
1. In the lectures we have tackled the *maximum segment sum* problem. We have derived a program from an executable, but inefficient specification. Can you use a similar approach to solve the maximum segment *product* problem?

   ```
   maximumSegmentProduct  ::  [Integer] → Integer
   maximumSegmentProduct  =  maximum ◦ map product ◦ segments
   ```

   What's the problem? Carefully study the derivation given in the lectures: which properties of maximum (`max`) and sum (`+`) are actually needed? Can you abstract away from these two operators?

2. The solution to the maximum segment sum problem that we derived in the lectures runs in linear time. However, it seems to be inherently sequential as the argument to `scanr` is not associative. Can you frame the problem as an instance of map-reduce so that it is easily parallelizable? Have a go! *Remember:* the art of map-reduce is to find a suitable monoid.

   *Hint:* apply the *inventor's paradox* solving a more complicated problem. To see what's needed, consider a division step (map-reduce is an instance of divide-and-conquer). There are three possible cases: the maximum segment sum lies entirely in the left half, entirely in the right half, or it crosses the dividing line.

   Thus, to calculate the maximum *segment* sum, we also require the maximum *prefix* sum and the maximum *suffix* sum. Now, consider computing the maximum prefix sum. There are only two cases: the maximum prefix lies entirely in the left half, or it crosses the dividing line.

   Thus, to compute the maximum prefix sum, we also require the overall sum, the total. Overall, we need to compute four values simultaneously:

(a) maximum segment sum;

(b) maximum prefix sum;

(c) maximum suffix sum;

(d) overall sum.

3. In the lectures we initially considered the *maximum profit problem*. The maximum segment sum problem was only the result of transforming the original problem. But is the transformation really necessary? How would you specify the original problem? Can you derive a program from the specification?

4. The original problem only allows for a *single* transaction: you buy one unit of stock and then sell it at a later point in time. Consider allowing multiple, *non-overlapping* transactions.

**Hints to practitioners 7.** Equational proofs and derivations shown in textbooks are often the result of a lot of polishing—like the proofs in the lectures. It has been said that proving 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. First apply obvious rewrites such as plugging in definitions. Perhaps you can identify an intermediate goal such as applying the induction assumption or e.g. Horner's rule. Try to systematically work towards this intermediate goal. When the proof is finalized, double check whether you have used all of the assumptions. If this is not the case, there is some chance that either the proof is wrong or the theorem is actually more general.