

Exercise sheet 2

Max Wisniewski, Alexander Steen

Task 1

Write the function

```
reverse []      = []
reverse (a:as) = reverse as ++ [a]
```

as `foldr` and `foldl`.

Solution:

The neutral element has to be `e = e' = []` because the empty list always returns the empty list.

In `foldr`

```
f a bs = {spec}
        bs ++ [a]
      = {shorten}
        flip (++) [a] bs
      ⇒
      f = flip (.) return (flip (++))
```

has to hold. Of course we could use the shortened version, but we tried it with a point free version. This function takes the `a` as a first argument. This one is bound after the first application of `(.)` by the `return` which will pack the `'a'` in a list. This one will be bound as the second argument in the `(++)`.

In `foldl`

```
      foldl f e (a:as)
= {spec foldl}
  foldl f (f e a) as

      f bs a
= {spec}
  a : bs
= {shorten}
  flip (:) bs a
```

holds. This spec holds, because `bs` is the already reversed first list. Therefore `'a'` is the element last to the yet seen list. We have to put it in front. We get the solution code

```
reverseR :: [a] → [a]
reverseR = foldr (flip (.) return (flip (++))) []
```

```
reverseL :: [a] → [a]
reverseL = foldl (flip (:)) []
```

Task 2

Write the function

```
concat :: [[a]] → [a]
concat []      = []
concat (as:ass) = as ++ concat ass
```

as a `foldl` and as a `foldr`.

Solution:

Again by definition $e = e' = []$ has to hold.

In `foldr`

```
f as bs      = {spec}
               as ++ bs
               <=>
f             = (++)
```

has to hold.

In `foldl`

```
foldl f e (as:ass)
= {def. foldl}
  foldl f (f e as) ass
= {*}
  foldl f (e ++ as) ass
⇒
  f = (++)
```

holds. Because in $(*)$ `bs` is the already concated list up to the list `as` and by the specification `as` has to be put at the end.

This leads to the functions

```
concatL :: [[a]] → [a]
concatL = foldl (++) []
```

```
concatR :: [[a]] → [a]
concatR = foldr (++) []
```

Task 3

Proof by induction on listst that `reverse ∘ reverse = id`.

Solution:

Lemma 1: `reverse (as ++ bs) = (reverse bs) ++ (reverse as)`.

Proof: Induction on `as`.

```
I.A. as = []
    reverse ([] ++ bs)
= {def. ++}
  reverse bs
= {def. ++}
  (reverse bs) ++ []
```

```

= {def. reverse}
  (reverse bs) ++ (reverse [])

I.S. as → (a:as)
    reverse ((a:as) ++ bs)
= {def. ++}
  reverse (a:(as ++ bs))
= {def. reverse}
  (reverse (as ++ bs)) ++ [a]
= {ind. hyp.}
  (reverse bs) ++ (reverse as) ++ [a]
= {++ assoziative}
  (reverse bs) ++ ((reverse as) ++ [a])
= {def. ++}
  (reverse bs) ++ (reverse (a:as))

```

□

Proof (main claim):

```

I.A. n = []
    reverse ( reverse [] )
= {def reverse.1}
  reverse []
= {def reverse.1}
  []
= {def. id}
  id []

I.S. n = (a:as)
    reverse ( reverse (a:as) )
= {def reverse.2}
  reverse ((reverse as) ++ [a])
= {lemma 1}
  reverse [a] ++ reverse (reverse as)
= {def. reverse x2}
  [a] ++ reverse (reverse as)
= {ind. hyp}
  [a] ++ id as
= {def. id}
  [a] ++ as
= {def. ++ x2}
  (a:as)
= {def. id}
  id (a:as)

```

□

Task 4

Formulate a Fusion law for `foldNat`, `foldSTree` and `foldPair`.

foldNat: This one is defined by

```
foldNat :: (a → a) → a → Nat → a
```

```

foldNat f e 0      = e
foldNat f e (S n)  = f (foldNat f e n)

```

We want to find a law of the form $h \circ \text{foldNat } f \ e = \text{foldNat } f' \ e'$.

We start with the base.

```

      h (foldNat f e 0)
= {def. foldNat}
  h e
= {spec, cond A}
  e'
= {def. foldNat}
  foldNat f' e' 0

```

We obtain, that $e' = h \ e$ has to hold.

```

      h (foldNat f e (S n))
= {def. foldNat}
  h (f (foldNat f e n))
= {cond B}
  f' (h (foldNat f e n))
= {ind. hyp}
  f' (foldNat f' e' n)
= {def. foldNat}
  foldNat f' e' (S n)

```

We obtain the condition, that $h \ (f \ a) = f' \ (h \ a)$

Theorem 1. (*FoldNat Fusion Law*)

Let $f : a \rightarrow a$ and $h : a \rightarrow b$ be functions and $e : a$ an object.

Then for any function $f' : b \rightarrow b$ with $h \ (f \ a) = f' \ (h \ a)$ and $e' = h \ e$ the following holds.

$$h \circ \text{foldNat } f \ e = \text{foldNat } f' \ e'$$

foldSTree: This one is defined by

```

foldSTree :: (a → b → a → a) → a → STree b → a
foldSTree f e Empty      = e
foldSTree f e (Node lt a rt) = f (foldSTree f e lt) a (foldSTree f e rt)

```

We want to find a law of the form

$$h \circ \text{foldSTree } f \ e = \text{foldSTree } f' \ e'$$

We begin with the base case.

```

      h ( foldSTree f e Empty )
= {def. foldSTree}
  h e
= {cond A}
  e'
= {def. foldSTree}
  foldSTree f' e' Empty

```

We proceed by induction

```

      h (foldSTree f e (Node lt a rt))
=   {def. foldSTree}
      h (f (foldSTree f e lt) a (foldSTree f e rt))
=   {cond B}
      f' (h (foldSTree f e lt)) a (h (foldSTree f e rt))
=   {ind. hyp}
      f' (foldSTree f' e' lt) a (foldSTree f' e' rt)
=   {def. foldSTree}
      foldSTree f' e' (Node lt a rt)

```

We obtained a second claim we need, that is $h (f \ l \ a \ r) = f' (h \ l) \ a \ (h \ r)$.

Theorem 2. (*FoldSTree Fusion Law*)

Let $f : a \rightarrow b \rightarrow a \rightarrow a$ and $h : a \rightarrow c$ be functions and $e : a$ an object.

Then for $f' : c \rightarrow b \rightarrow c \rightarrow c$ with $f' (h \ l) \ a \ (h \ r) = h (f \ l \ a \ r)$ and $e' = h \ e$ the following holds.

$$h \circ \text{foldSTree } f \ e = \text{foldSTree } f' \ e'$$

foldPair: This one is defined by

```

foldPair :: (a → b → c) → (a,b) → c
foldPair f (a,b) = f a b

```

We want to find a law of the form

$$h \circ \text{foldPair } f = \text{foldPair } f'$$

We begin on the left side and because we have only one case may start with it.

```

      h ∘ foldPair f (a,b)
=   {def. foldPair}
      h ( f a b )
=   {spec}
      f' a b
=   {def. foldPair}
      foldPair f' (a,b)

```

We obtain the following law.

Theorem 3. (*FoldPair Fusion Law*)

Let $f : a \rightarrow b \rightarrow c$ be a function and $h : c \rightarrow d$ be a function. Then for $f' \ a \ b = h (f \ a \ b)$ the following hold

$$h \circ \text{foldPair } f = \text{foldPair } f'$$