$\boxed{9.1}$

α ←

```
data Nat = Zero | Succ Nat
```

The function int :: Nat -> Int transforms an argument of type Nat into an Int:

```
int :: Nat -> Int
int Zero = 0
int (Succ x) = 1 + int x
```

The function nat :: Int -> Nat transforms an Int into a result of type Nat.

```
nat :: Int -> Nat
nat 0 = Zero
nat x = Succ (nat (x-1))
```

Now, the functions add, mul, pow, tet:

```
add :: Nat -> Nat -> Nat
add x Zero = x
add x (Succ y) = Succ (add x y)

mul :: Nat -> Nat -> Nat
mul _ Zero = Zero
mul x (Succ y) = add x (mul x y)

pow :: Nat -> Nat -> Nat
pow _ Zero = Succ Zero
pow x (Succ y) = mul x (pow x y)

tet :: Nat -> Nat -> Nat
tet _ Zero = Succ Zero
tet x (Succ y) = pow x (tet x y)
```

$\boxed{9.2}$ For lists, we had foldr f e [] = e and foldr f e (x:xs) = f x (foldr f e xs). Applying the same reasoning for the Nat type (whose constructors are Succ and Zero), we get:

```
foldNat :: (a -> a) -> a -> Nat -> a
foldNat f e Zero = e
foldNat f e (Succ x) = f (foldNat f e x)
```

This fold instance has 2 properties: when we have foldNat (f e) applied to Zero, it returns the value of the accumulator, which is e, and when applied to a Succ x, it returns the value of f applied to foldNat f e x (until we get to Zero), so this is a recursive procedure. In the case of lists we have foldr (:) [] = id, as (:) and [] are the constructors for lists, whereas here we have foldNat Succ Zero = id, as Succ and Zero are the constructors for the Nat type (here id :: Nat -> Nat).

The deconstructors of Nat are :

- the DISCRIMINATORS:

```
isZero :: Nat -> Bool            isSucc :: Nat -> Bool
isZero Zero = True               isSucc Zero = False
isZero _ = False                 isSucc _ = True
```

which tells if the argument we use is Zero or Succ Nat.

- the SELECTORS:

bad name!

```
zero :: Nat -> Nat               succ' :: Nat -> Nat
zero Zero = Zero                 succ' (Succ x) = x -- succ is already defined in Haskell
```

which are defined only partially (zero only for Zero, succ' only for non-Zero arguments).

α - Now, we'll define unfoldNat:

```
unfoldNat :: (a -> Bool) -> (Nat -> Nat) -> (a -> a) -> a -> Nat
unfoldNat done first next x
    | done x = Zero
    | otherwise = first (unfoldNat done first next (next x))
```

This was expected to be Succ

The function unfoldNat is characterised by the fact that it returns the result of first, applied to Zero n times, where n is the number of times we apply next to x until done x becomes true. The identity case for unfoldNat is

unfoldNat isZero Succ succ' = id, where id :: Nat -> Nat.

Now, let's express int, nat, add, mul, pow, tet with foldNat or unfoldNat:

```
int :: Nat -> Int                nat :: Int -> Nat
int = foldNat (+1) 0             nat = unfoldNat (==0) Succ pred

add :: Nat -> Nat                mul :: Nat -> Nat
add x = foldNat Succ x           mul x = foldNat (add x) Zero

pow :: Nat -> Nat                tet :: Nat -> Nat
pow x = foldNat (mul x) (Succ Zero)   tet x = foldNat (pow x) (Succ Zero)
```

10.1 We want to prove by induction on xs that

(*) $fold\ c\ n\ (xs +\!\!+ ys) = fold\ c\ (fold\ c\ n\ ys)\ xs$, for all lists xs and ys (whether partial, finite or infinite).

First of all, we recall that:

$fold :: (a \to b \to b) \to b \to [a] \to b$

$fold\ c\ n\ [\,] = n$

$fold\ c\ n\ (x : xs) = c\ x\ (fold\ c\ n\ xs)$

and

$(+\!\!+) :: [a] \to [a] \to [a]$

$[\,] +\!\!+ ys = ys$

$(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)$

Ⓐ Induction over FINITE LISTS

1. $P([\,])$: $fold\ c\ n\ ([\,] +\!\!+ ys) =$
$= \{\text{definition of } (+\!\!+)\}$
$\underline{fold\ c\ n\ ys}$

and

$fold\ c\ (fold\ c\ n\ ys)\ [\,] =$
$= \{\text{definition of fold}\}$
$\underline{fold\ c\ n\ ys}$

2. For every finite lists xs, if P(xs) then P(x:xs)

$P(x:xs)$: $fold\ c\ n\ ((x:xs) +\!\!+ ys) =$
$= \{\text{definition of } (+\!\!+)\}$
$fold\ c\ n\ (x : (xs +\!\!+ ys)) =$
$= \{\text{definition of fold}\}$
$c\ x\ (fold\ c\ n\ (xs +\!\!+ ys)) =$
$= \{\text{induction hypothesis}\}$
$c\ x\ (fold\ c\ (fold\ c\ n\ ys)\ xs) =$
$= \{\text{definition of fold}\}$
$fold\ c\ (fold\ c\ n\ ys)\ (x : xs)$

Therefore, we proved (*) for all finite lists.

Ⓑ Induction over PARTIAL LISTS

1. $P(\perp)$: $fold\ c\ n\ (\perp +\!\!+ ys) =$
$= \{(+\!\!+) \text{ is strict in its left argument}\}$
$fold\ c\ n\ \perp$
$= \{\text{fold is strict in its third argument}\}$
$\perp$

$fold\ c\ (fold\ c\ n\ ys)\ \perp =$
$= \{\text{fold is strict in its third argument}\}$
$\perp$

2. For every partial list xs, if $P(xs)$ then $P(x:xs)$
Here, we use the same reasoning as we did at Ⓐ2. and we get the same result.
Therefore, we proved ⊛ for all partial lists.

Ⓒ <u>INFINITE LISTS</u>

As ⊛ holds for all finite and partial lists, and since it is an equation between Haskell expressions it is chain complete and also holds for infinite lists.

$\boxed{10.2}$ First, we want to prove that $(\mathbin{+\!\!+} bs)$ is a fold using fold fusion:

$(\mathbin{+\!\!+} bs) =$
$= \{\text{composing with id}\}$
$(\mathbin{+\!\!+} bs) . id =$
$= \{fold\ (:)\ [\,] = id\}$
$\underline{(\mathbin{+\!\!+} bs) . fold\ (:)\ [\,]}$

$(\mathbin{+\!\!+} bs) =$
$= \{\text{writing it as a fold}\}$
$\underline{fold\ h\ a}$

You've missed the point here: you're supposed to use fusion, not induction,

Now, we study these two applied to $\perp$, $[\,]$, $(x:xs)$, where the inductive hypothesis holds for xs.

1. $((\mathbin{+\!\!+} bs) . fold\ (:)\ [\,])\perp =$
$= \{\text{definition of composition}\}$
$(\mathbin{+\!\!+} bs)(fold\ (:)\ [\,]\ \perp) =$
$= \{\text{the strictness of fold}\}$
$(\mathbin{+\!\!+} bs)\ \perp =$
$= \{\text{definition of } (\mathbin{+\!\!+} bs)\}$
$\perp \mathbin{+\!\!+} bs =$
$= \{\text{strictness (in the } 1^{st} \text{ arg.) of } (\mathbin{+\!\!+})\}$
$\perp$

$fold\ h\ a\ \perp =$
$= \{\text{strictness of fold}\}$
$\perp$

So, this equality holds anyways.

2. $((\mathbin{+\!\!+} bs) . fold\ (:)\ [\,])\ [\,] =$
$= \{\text{definition of composition}\}$
$(\mathbin{+\!\!+} bs)(fold\ (:)\ [\,]\ [\,]) =$
$= \{\text{definition of fold}\}$
$(\mathbin{+\!\!+} bs)\ [\,] =$
$= \{\text{definition of } (\mathbin{+\!\!+} bs)\}$
$[\,] \mathbin{+\!\!+} bs =$
$= \{\text{definition of } (\mathbin{+\!\!+})\}$
bs

$fold\ h\ a\ [\,] =$
$= \{\text{definition of fold}\}$
a

So, from the equality for [], we need $a = bs$.

3. $((+\!\!+ bs) . \text{fold } (:)\ [\,])\ (x:xs) =$
$= \{$definition of composition$\}$
$(+\!\!+ bs)\ (\text{fold } (:)\ [\,]\ (x:xs)) =$
$= \{\text{fold } (:)\ [\,] = id\}$
$(+\!\!+ bs)\ (x:xs) =$
$= \{$definition of $(+\!\!+ bs)\}$
$(x:xs) +\!\!+ bs =$
$= \{$definition of $(+\!\!+)\}$
$x : (xs +\!\!+ bs)$

$f\ h\ a\ (x:xs) =$
$= \{$definition of fold$\}$
$h\ x\ (\text{fold } h\ a\ xs) =$
$= \{$inductive hypothesis$\}$
$h\ x\ (((+\!\!+ bs) . \text{fold } (:)\ [\,])\ xs) =$
$= \{$definition of composition$\}$
$h\ x\ ((+\!\!+ bs)(\text{fold } (:)\ [\,]\ xs)) =$
$= \{\text{fold } (:)\ [\,] = id\}$
$h\ x\ ((+\!\!+ bs)\ xs) =$
$= \{$definition of $(+\!\!+ bs)\}$
$(h\ x)\ (xs +\!\!+ bs)$

From the equality, we get
$h\ x = (x:)$, so $h = (:)$.

From 1, 2 and 3 we conclude that:
$(+\!\!+ bs) = \text{fold } (:)\ bs$.

Now, we want to deduce without resort to induction that
(*) $\text{fold } c\ m\ (xs +\!\!+ ys) = \text{fold } c\ (\text{fold } c\ m\ ys)\ xs$
We already proved that $(+\!\!+ bs) = \text{fold } (:)\ bs$ and therefore we'll replace $xs +\!\!+ ys$ with $(+\!\!+ ys)\ xs$ and then with $\text{fold } (:)\ ys\ xs$
So, we basically need to prove that
$\text{fold } c\ m\ (\text{fold } (:)\ ys\ xs) = \text{fold } c\ (\text{fold } c\ m\ ys)\ xs$
$\boxed{(\text{fold } c\ m) . (\text{fold } (:)\ ys) = \text{fold } c\ (\text{fold } c\ m\ ys)}$ (**)
Now, let $f = \text{fold } c\ m$
$g = (:)$
$a = ys$
$h = c$
$b = \text{fold } c\ m\ ys$
We want to prove that

$f . (\text{fold } g\ a) = \text{fold } h\ b$, which is the fold fusion for f. For this to be true, it needs to have 3 properties:

1) <u>f must be strict</u>, but since $f = \text{fold } c\ m$, which is strict $(\text{fold } c\ m\ \bot = \bot)$, this is true

2) $b = f\ a$, where $b =$ fold c m ys, $f =$ fold c m, $a = ys$ $\Rightarrow b = f\ a$, so this is also true ✓

3) $h\ x . f = f . g\ x$

(c x). (fold c m) = (fold c m). ((:) x), or

((c x). (fold c m)) ys = ((fold c m). ((:) x)) ys

{definition of composition}

(c x) (fold c m ys) = (fold c m) ((:) x ys)

{definition of fold + definition of (:)}

fold c m (x:ys) = fold c m (x:ys), which is true ✓

So, we proved (**), therefore (*) must be also true.

$\boxed{10.3}$ We want to use fold fusion to show that filter is a fold.

filter p =

={composing with id = fold (:) []}

(filter p). (fold (:) [])

filter p =

= {expressing it with fold}

fold h b

We recall that:

filter :: (a -> Bool) -> [a] -> [a]

filter _ [] = []

filter p (x:xs) = [x | p x] ++ filter p xs

Now, for the fold fusion, we need to have 3 properties:

1) ((filter p). (fold (:) [])) $\perp$

= {definition of composition}

(filter p) (fold (:) [] $\perp$) =

= {strictness of fold}

filter p $\perp$ =

= {strictness of filter}

$\perp$

fold h b $\perp$ =

= {strictness of fold}

$\perp$

So, this property is true.

Again, you should be using fusion here.

2) $((\text{filter } p) . (\text{fold } (:) \; [\,])) \; [\,] =$

$= \{\text{definition of composition}\}$

$(\text{filter } p) \, (\text{fold } (:) \; [\,] \; [\,]) =$

$= \{\text{definition of fold}\}$

$\text{filter } p \; [\,] =$

$= \{\text{definition of filter}\}$

$\underline{[\,]}$

Therefore, $b = [\,]$.

$\text{fold } h \; b \; [\,] =$

$= \{\text{definition of fold}\}$

$\underline{b}$

3) $\underbrace{(\text{filter } p)}_{f} . (\text{fold } \underset{g}{(:)} \; [\,]) = \text{fold } \underset{h}{h} \; b \; [\,]$

$h \; x . f = f \cdot g \; x$

$(h \; x) . (\text{filter } p) = (\text{filter } p) . ((:) \; x)$, therefore, it must be true for all arguments ys

$(h \; x) . (\text{filter } p) \; ys =$

$= \{\text{definition of composition}\}$

$\underline{(h \; x) \, (\text{filter } p \; ys)}$

$((\text{filter } p) . ((:) \; x)) \; ys =$

$= \{\text{definition of composition}\}$

$(\text{filter } p) . ((:) \; x \; ys) =$

$= \{\text{definition of } (:)\}$

$\text{filter } p \; (x : ys) =$

$= \{\text{definition of filter}\}$

$\underline{[x \mid p \; x] +\!\!+ \; \text{filter } p \; ys}$

Therefore, $h \; x = \text{if } p \; x \text{ then } ([x] +\!\!+) \text{ else } ([\,] +\!\!+)$ ✓

So, $\text{filter } p = \text{fold } h \; [\,]$, where h is defined above.

Now, we want to deduce that

$\text{filter } p \; (xs +\!\!+ ys) = \text{filter } p \; xs +\!\!+ \text{filter } p \; ys$.

First, we'll define h x in a similar way, but more explicit:

```
h :: a -> [a] -> [a]
h x xs = if (p x) then x : xs
                  else xs
```

Now, we start from the right side:

$$\text{filter } p\ xs \mathbin{+\!\!+} \text{filter } p\ ys =$$

$= \{$ definition of $(\mathbin{+\!\!+}\, bs)\}$

$$(\mathbin{+\!\!+}\ \text{filter } p\ ys)\ (\text{filter } p\ xs) =$$

$= \{$ writing filter p as a fold $\}$

$$(\mathbin{+\!\!+}\ \text{filter } p\ ys)\ (\text{fold } h\ []\ xs) =$$

$= \{$ definition of composition $\}$

$$((\mathbin{+\!\!+}\ \text{filter } p\ ys) . (\text{fold } h\ []))\ xs$$

Now, we'll use fold fusion for $(\mathbin{+\!\!+}\ \text{filter } p\ ys) . (\text{fold } h\ [])$

$$\underbrace{(\mathbin{+\!\!+}\ \text{filter } p\ ys)}_{f} . (\text{fold } \underset{g}{\underset{\downarrow}{h}}\ \underset{a}{\underset{\downarrow}{[]}}) = \text{fold } h'\ b'$$

1) f must be strict

$$(\mathbin{+\!\!+}\ \text{filter } p\ ys)\ \bot =$$

$= \{$ definition of $(\mathbin{+\!\!+}\, bs)\}$

$$\bot \mathbin{+\!\!+} \text{filter } p\ ys =$$

$= \{$ strictness of $(\mathbin{+\!\!+})\}$ ✓

$$\bot$$

2) $b' = f\ a \Rightarrow b' = (\mathbin{+\!\!+}\ \text{filter } p\ ys)\ [] =$

$= \{$ definition of $(\mathbin{+\!\!+}\, bs)\}$

$$[] \mathbin{+\!\!+} \text{filter } p\ ys =$$

$= \{$ definition of $(\mathbin{+\!\!+})\}$

$$\text{filter } p\ ys$$ ✓

So, $\underline{b' = \text{filter } p\ ys}$

3) $h'\ x . f = f . g\ x$ (We apply them to the same argument xs)

$$((h'\ x) . (\mathbin{+\!\!+}\ \text{filter } p\ ys))\ xs =$$

$= \{$ definition of composition $\}$

$$(h'\ x)((\mathbin{+\!\!+}\ \text{filter } p\ ys)\ xs) =$$

$= \{$ definition of $(\mathbin{+\!\!+}\, bs)\}$

$$(h'\ x)\ (xs \mathbin{+\!\!+} \text{filter } p\ ys)$$

And now the RHS:

$$((+\!\!+\ \text{filter}\ p\ ys) . (h\ x))\ xs =$$

$= \{$definition of composition$\}$

$$(+\!\!+\ \text{filter}\ p\ ys)\ (h\ x\ xs) \underset{\substack{\uparrow \\ \text{using the definition of } h}}{=} \begin{cases} \text{if } p\ x \text{ then } (+\!\!+\ \text{filter}\ p\ ys)\ (x:xs) \\ \text{else } (+\!\!+\ \text{filter}\ p\ ys)\ xs \end{cases}$$

That means that (using the definition of $(+\!\!+\ bs)$):

$$(+\!\!+\ \text{filter}\ p\ ys)(h\ x\ xs) = \begin{cases} \text{if } p\ x \text{ then } (x:xs) +\!\!+ (\text{filter}\ p\ ys) \overset{\substack{\text{using the definition of } (+\!\!+) \\ \downarrow}}{=} x : (xs +\!\!+ \text{filter}\ p\ ys) \\ \text{else } xs +\!\!+ \text{filter}\ p\ ys \end{cases}$$

So, again by using the definition of h:

$$(+\!\!+\ \text{filter}\ p\ ys)\ (h\ x\ xs) = h\ x\ (\text{filter}\ p\ ys) = (h\ x)(\text{filter}\ p\ ys)$$

$$LHS = RHS \Rightarrow (h'\ x)(\text{filter}\ p\ ys) = (h\ x)(\text{filter}\ p\ ys) \Rightarrow \underline{h' = h}$$ ✓

Therefore, from the fold fusion we get:

$(+\!\!+\ \text{filter}\ p\ ys) . (\text{fold}\ h\ [\,]) = \text{fold}\ h\ (\text{filter}\ p\ ys)$ and, as we started from $\text{filter}\ p\ xs +\!\!+ \text{filter}\ p\ ys \Rightarrow \boxed{\text{filter}\ p\ xs +\!\!+ \text{filter}\ p\ ys = \text{fold}\ h\ (\text{filter}\ p\ ys)\ xs}$ (A)

Now, working on the left side:

$\underline{\text{filter}\ p\ (xs +\!\!+ ys)} =$

$= \{$definition of $(+\!\!+\ bs)\}$

$\text{filter}\ p\ ((+\!\!+\ ys)\ xs) =$

$= \{$expressing $(+\!\!+\ bs)$ as a fold from $\boxed{10.2}\}$

$\text{filter}\ p\ (\text{fold}\ (:)\ ys\ xs) =$

$= \{$expressing filter p as a fold$\}$

$(\text{fold}\ h\ [\,])(\text{fold}\ (:)\ ys\ xs) =$

$= \{$definition of composition$\}$

$((\text{fold}\ h\ [\,]) . (\text{fold}\ (:)\ ys))\ xs =$

$= \{$(**) equality from page 5.$\}$ or using Q10.1

$\text{fold}\ h\ (\text{fold}\ h\ [\,]\ ys)\ xs =$

$= \{$expressing filter p as a fold$\}$

$\underline{\text{fold}\ h\ (\text{filter}\ p)\ xs}$

Therefore, we obtained that $\boxed{\text{filter}\ p\ (xs +\!\!+ ys) = \text{fold}\ h\ (\text{filter}\ p)\ xs}$ (B) ✓

From (A) and (B) we finally deduce that $\text{filter}\ p\ (xs +\!\!+ ys) = \text{filter}\ p\ xs +\!\!+ \text{filter}\ p\ ys$.

10.4 α −

```
data Liste a = Snoc (Liste a) a | Lim deriving (Show, Eq)

cat :: Liste a -> Liste a -> Liste a
cat xs Lim = xs
cat xs (Snoc ys y) = Snoc (cat xs ys) y
```

Basically, here we replace the Lim from ys with xs, as that's the concatenation for the Liste a type.

```
folde :: (b -> a -> b) -> b -> Liste a -> b
folde f lin Lim = lin
folde f lin (Snoc xs x) = f (folde f lin xs) x
```

We notice that, as expected, Lim and Snoc are the constructors for the Liste a type, id = folde Snoc Lim. We can also notice that the structure of folde is similar to the structure of foldl.

Now, we express cat in terms of folde:

```
cat' :: Liste a -> Liste a -> Liste a
cat' xs = folde Snoc xs
```

As we cannot express liste with folde because of the types, we use fold, which is defined as:

```
list :: Liste a -> [a]
list = folde (\xs x -> xs ++ [x]) [] -- the lambda function adds elements at the
                                     -- end of the result list

fold :: (a -> b -> b) -> b -> [a] -> b
fold cons nil [] = nil
fold cons nil (x:xs) = cons x (fold cons nil xs)

liste :: [a] -> Liste a
liste = fold (\x xs -> cat' (Snoc Lim x) xs) Lim
```

In our case, the cons parameter is the lambda function, which, when given an element x and a Listea(xs), concatenates the Liste which only has x (Snoc Lin x) with xs, so that at the end we have the Liste result in a correct order. So, the list [a,b,c,d] will be (Snoc Lim a) 'cat' (Snoc Lin b) 'cat' (Snoc Lin c) . . .

Now, if we have an infinite list, let's say [1..], then liste will have to calculat (Snoc Lim 1) 'cat' liste [2..], then (Snoc Lim 1) 'cat' (Snoc Lin 2) 'cat' liste [3..] and so on (we'll never reach a result). i.e. $\perp$.

The infinite objects of type Liste a are

Smoc ... ; Smoc (Smoc ...) a ; Smoc (Smoc (Smoc ...) b) a ...

We'll recall tailfold and define tailfolde (if the natural fold for Liste a is a foldl then tailfolde will have the structure of a foldr):

```
tailfold :: (b -> a -> b) -> b -> [a] -> b
tailfold cons nil [] = nil
tailfold cons nil (x:xs) = tailfold cons (cons nil x) xs
```

```
tailfolde :: (a -> b -> b) -> b -> Liste a -> b
tailfolde cons nil Lin = nil
tailfolde cons nil (Smoc xs x) = cons x (tailfolde cons nil xs)
-- id = tailfolde (flip Smoc) Lin , as Smoc :: Liste a -> a -> Liste a
```

This isn't tailfolde.

Now, we'll create list' and liste' with tailfolde and tailfold, respectively (because of type restrictions):

```
list' :: Liste a -> [a]
list' = tailfolde (\x xs -> xs ++ [x]) []
```

(Ok with your definition)

Similar to list, but with the lambda function flipped, to respect the type restrictions of tailfolde.

```
liste' :: [a] -> Liste a
liste' = tailfold Smoc Lin
```

$\boxed{10.5}$ First, we recall the unfold function for [a]

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
unfold n h t x
  | n x = []
  | otherwise = h x : unfold n h t (t x)
```

which yields to identity, when n = null, h = head, t = tail (deconstructors for [a])

Now, we define unfolde for Liste a :

```
unfolde :: (b -> Bool) -> (b -> b) -> (b -> a) -> b -> Liste a
unfolde n h t x
  | n x = Lin
  | otherwise = Smoc (unfolde n h t (h x)) (t x)
```

Misleading names

Here, for identity, we need n h t to be the deconstructors for Liste a which are (== Lin), (\(Smoc xs x) -> xs) and (\(Smoc xs x) -> x), respectively.

For us to define list" and liste" using unfolds, we first need to create the equivalent of init :: [a] -> [a] and last :: [a] -> a, for Liste a, and we'll do that recursively:

```
inite :: Liste a -> a
inite (Snoc Lin x) = x
inite (Snoc xs x) = inite xs
```

Bad name: call this heade

xs = Snoc (Snoc (... (Snoc Lin a_1) a_2) ...) a_m , inite Lin is not defined, as last [] is not defined either.

↑ inite xs

α (−)

```
laste :: Liste a -> Liste a
laste (Snoc Lin x) = Lin
laste (Snoc xs x) = Snoc (laste xs) x
```

Again a bad name: call this taile.

xs = Snoc (Snoc (... (Snoc Lin a_1) a_2) ...) a_n
laste xs = Snoc (Snoc (... (Snoc Lin a_2) a_3) ...) a_n
laste Lin is not defined, as init [] is not defined either.

Now, we can write list" and liste" using unfold and unfolde, respectively.

need this for the checker (==Lin)

```
list" :: Eq a => Liste a -> [a]
list" = unfold (== Lin) inite laste
```

better to define
isLin Lin = True
isLin _ = False.

```
liste" :: [a] -> Liste a
liste" = unfolde null init last
```