Construction/Déconstruction

```
{-# LANGUAGE DeriveFunctor #-}
data Fix f = Fx (f (Fix f))
unFix :: Fix f -> f (Fix f)
unFix (Fx x) = x;
data F a = Const Int | Plus a a deriving (Show,Functor)
calc :: F Int -> Int
calc (Const i) = i
calc (Plus i j) = i + j
cata :: Functor f => (f b -> b) -> Fix f -> b
cata g = g . (fmap (cata g)) . unFix
main=do print $ (cata calc) (Fx (Const 1))
 print $ (cata calc) $ Fx $ (Fx (Const 1)) `Plus` (Fx (Const 1))
```

$$\begin{tikzcd}
f\ (\mathrm{Fix}\ f) \arrow[r, "\mathrm{fmap}\ (\mathrm{cata}\ g)"] \arrow[d, shift left, "\mathrm{Fix}"] & F\ a \arrow[d] \\
\mathrm{Fix}\ f \arrow[u, shift left, "\mathrm{unFix}"] \arrow[r, "\mathrm{cata}\ g"'] & a
\end{tikzcd}$$

```haskell
{-# LANGUAGE DeriveFunctor #-}
-- https://bartoszmilewski.com/2017/02/28/f-algebras/
import Prelude
newtype Fix f = Fix (f (Fix f))
unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix;
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
data StreamF e a = StreamF e a deriving (Functor,Show)
al :: StreamF e [e] -> [e]
al (StreamF e a) = e : a
toListC :: Fix (StreamF e) -> [e]
toListC = cata al
notdiv p n = n `mod` p /= 0
erat :: [Int] -> StreamF Int [Int]
erat (p : ns) = StreamF p (filter (notdiv p) ns)
main = do print $ (toListC . (ana erat)) [2..]
```

$$\begin{tikzcd}
F\ a \arrow[r, "fmap\ (ana\ g)"] & f\ (Fix\ f) \arrow[d, shift right, "Fix"'] \\
a \arrow[u, "g"] \arrow[r, "ana\ g"] & Fix\ f \arrow[u, shift right, "u"']
\end{tikzcd}$$

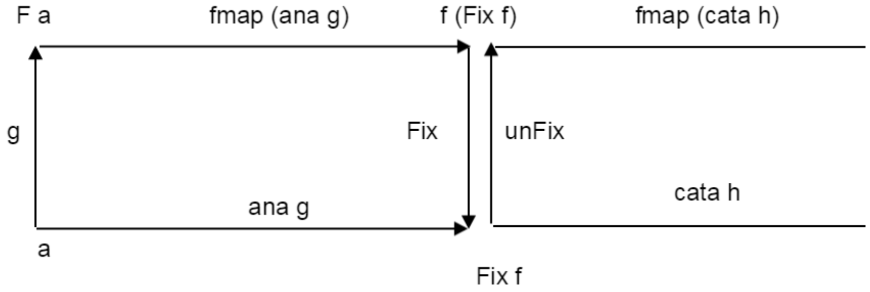
Crible d'Ératosthène

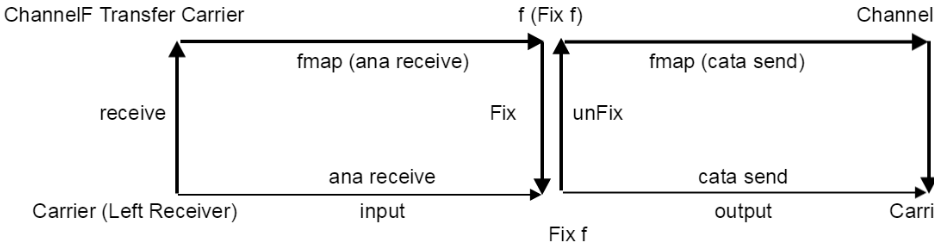
https://fr.wikipedia.org/wiki/Crible_d%27%C3%89ratosth%C3%A8ne

Main.hs	module Main where
	import Prelude
	import Data.Ampli.Ampli
	main = do print $ ampli (Left "h1 e2")

Ampli.hs

```
{-# LANGUAGE DeriveFunctor #-}
module Data.Ampli.Ampli where
import Prelude
import Data.Ampli.Hylo
data StreamF e a = StreamF e a
deriving (Functor,Show)
data ChannelF e a = NilF |
ChannelF e a deriving (Functor,
Show)
type Carrier = Either Receiver
Sender
type Transfer = Maybe Int
type ConnectorF = ChannelF
Transfer
type InterfaceF = ConnectorF
Carrier
output :: Fix (ConnectorF) ->
Carrier
input :: Carrier -> Fix
(ConnectorF)
output = cata send
input = ana receive
ampli = (output . input)
type Receiver = [Char]
type Sender = [Int]
send:: InterfaceF -> Carrier
receive:: Carrier -> InterfaceF
send (ChannelF Nothing (Right p))
= (Right p)
send (ChannelF (Just i) (Right
p)) = (Right (i:p))
send (NilF) = (Right [])
receive (Left []) = NilF
receive (Left ('e':'1':ns)) =
ChannelF (Just 48) (Left ns)
receive (Left ('e':'2':ns)) =
ChannelF (Just 49) (Left ns)
receive (Left (p :ns)) = ChannelF
(Nothing) (Left ns)
```





Hylo.hs	{-# LANGUAGE DeriveFunctor #-} module Data.Ampli.Hylo where import Prelude data Fix f = Fix (f (Fix f)) instance Show (Fix f) where show (Fix x)= "." unFix :: Fix f -> f (Fix f) unFix (Fix x) = x cata :: Functor f => (f a -> a) -> Fix f -> a cata alg = alg . fmap (cata alg) . unFix; ana :: Functor f => (a -> f a) -> a -> Fix f ana coalg = Fix . fmap (ana coalg) . coalg