```
parEvalN :: [a -> b] -> [a] -> [b]
parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
parEvalN fs as = zipWith ($) fs as ‘using‘ parList rdeepseq
parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
parEvalN fs as = runPar $
(sequenceA $ map (spawnP) $ zipWith ($) fs as) >>= mapM get
```

```
parEvalN :: (Trans a, Trans b) => [a -> b] -> [a] -> [b]
parEvalN fs as = spawnF fs as
```

```
class Arrow arr where
  arr   :: (a -> b) -> arr a b
  (>>>) :: arr a b -> arr b c -> arr a c
  first :: arr a b -> arr (a,c) (b,c)
```

Functions (->) are arrows:

```haskell
instance Arrow (->) where
 arr  f  = f
 f >>> g = g . f
  first   f  = \(a, c)  -> (f a, c)
```

The Kleisli type

```
data Kleisli m a b = Kleisli { run :: a -> m b }
```

as well:

```
instance Monad m => Arrow (Kleisli m) where
 arr f = Kleisli $ return . f
 f >>> g = Kleisli $ \a -> f a >>= g
  first f = Kleisli $ \(a,c) -> f a >>= \b -> return (b,c)
```

```
second :: Arrow arr => arr a b -> arr (c, a) (c, b)
second f = arr swap >>> first f >>> arr swap
where swap (x, y) = (y, x)
```

```
(***) :: Arrow arr => arr a b -> arr c d -> arr (a, c) (b, d)
f *** g = first f >>> second g
```

```
(&&&) :: Arrow arr => arr a b -> arr a c -> a a (b, c)
f &&& g = arr (\a -> (a, a)) >>> (f *** g)
```

Arrow usage Example

```haskell
add :: Arrow arr => arr a Int -> arr a Int -> arr a Int
add f g = (f &&& g) >>> arr (\(u, v) -> u + v)
```

mapArr

The mapArr combinator lifts any arrow arr a b to an arrow arr [a] [b] [1],

```
mapArr :: ArrowChoice arr => arr a b -> arr [a] [b]
mapArr f =
   arr listcase >>>
   arr (const []) ||| (f *** mapArr f >>> arr (uncurry (:)))
   where
      listcase [] = Left ()
      listcase (x:xs) = Right (x,xs)
```

with

```
(|||) :: ArrowChoice arr a c -> arr b c -> arr (Either a b) c
```

zipWithArr

zipWithArr lifts any arrow `arr (a, b) c` to an arrow `arr ([a], [b]) [c]`.

```
zipWithArr :: ArrowChoice arr => arr (a, b) c -> arr ([a], [b]) [c]
zipWithArr f = (arr $ \(as, bs) -> zipWith (,) as bs) >>> mapArr f
```

```
listApp :: (ArrowChoice arr, ArrowApply arr) =>
  [arr a b] -> arr [a] [b]
listApp fs = (arr $ \as -> (fs, as)) >>> zipWithArr app
```

This combinator also makes use of the ArrowApply typeclass which allows us to evaluate arrows with `app :: arr (arr a b, a) c`.

[1] John Hughes. *Programming with Arrows*, pages 73–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31872-9. doi: 10.1007/11546382_2. URL `http://dx.doi.org/10.1007/11546382_2`.