```java
public static int fib(int x) {
  if (x<=0)
    return 0;
  else if (x==1)
    return 1;
  else
    return fib(x-2) + fib(x-1);
}
```

```haskell
fib :: Int -> Int
fib x
   | x <= 0 = 0
   | x == 1 = 0
   | otherwise =
     (fib (x - 2))
       + (fib (x - 1))
```

- Functional programming equally powerful as imperative programming
- focused on the "what?" instead of the "how?"
$\Rightarrow$ more concise $\Rightarrow$ easier to reason about
- based on Lambda Calculus

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Similar to Java's Optional, we have **Maybe** a:

```haskell
instance Monad Maybe where
  (Just a) >>= f = f a
  Nothing >>= _ = Nothing
  return a = Just a
```

$\Rightarrow$ composable computation descriptions

With monadic functions like

```haskell
func :: Int -> Maybe Int
func x
  | x < 0 = Nothing
  | otherwise = Just (x * 2)
```

we can compose computations:

```haskell
complicatedFunc :: Int -> Maybe Int
complicatedFunc x = (return x) >>= func >>= ...
```

Another way to compose computations are arrows:

```haskell
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 }
```

is also an arrow:

```
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)
```

Some Combinators:

```
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:

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

In general, Parallelism can be looked at as:

```
parEvalN :: [a -> b] -> [a] -> [b]
```

```haskell
parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
parEvalN fs as = zipWith ($) fs as `using` parList rdeepseq
```

with

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
using :: a -> Strategy a -> a
parList :: Strategy a -> Strategy [a]
rdeepseq :: NFData a => Strategy a
```

```
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
```

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 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 converts a list of arrows `[arr a b]` to a new arrow `arr [a] [b]`:

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

with the ArrowApply that defines `app :: arr (arr a b, a) c`.

```
parEvalN :: [a -> b] -> [a] -> [b]
```

```
parEvalN :: (Arrow arr) => [arr a b] -> arr [a] [b]
```

The ArrowParallel typeclass

```haskell
class Arrow arr => ArrowParallel arr a b where
  parEvalN :: [arr a b] -> arr [a] [b]
```

```haskell
class Arrow arr => ArrowParallel arr a b conf where
  parEvalN :: conf -> [arr a b] -> arr [a] [b]
```

```haskell
instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
  ArrowParallel arr a b conf where
    parEvalN _ fs = listApp fs >>> arr (flip using $ parList rdeepseq)
```

```haskell
instance (NFData b, ArrowApply arr, ArrowChoice arr) =>
  ArrowParallel arr a b conf where
    parEvalN _ fs =
      (arr $ \as -> (fs, as)) >>>
      zipWithArr (app >>> arr spawnP) >>>
      arr sequenceA >>>
      arr (>>= mapM get) >>>
      arr runPar
```

```haskell
instance (Trans a, Trans b) => ArrowParallel (->) a b conf where
parEvalN _ fs as = spawnF fs as
```

and the Kleisli type.

```haskell
instance (Monad m, Trans a, Trans b, Trans (m b)) =>
  ArrowParallel (Kleisli m) a b conf where
parEvalN conf fs =
  (arr $ parEvalN conf (map (\(Kleisli f) -> f) fs)) >>>
  (Kleisli $ sequence)
```

```haskell
class (Arrow arr) => ArrowUnwrap arr where
 arr a b -> (a -> b)
```

With the ArrowParallel typeclass in place and implemented, we can now implement some basic parallel skeletons.

```
parEvalNLazy :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply a
  conf -> ChunkSize -> [arr a b] -> (arr [a] [b])
parEvalNLazy conf chunkSize fs =
   arr (chunksOf chunkSize) >>>
   listApp fchunks >>>
   arr concat
  where
     fchunks = map (parEvalN conf) $ chunksOf chunkSize fs
```

```haskell
arrMaybe :: (ArrowApply arr) => (arr a b) -> arr (Maybe a) (Maybe b)
arrMaybe fn = (arr $ go) >>> app
  where
    go Nothing = (arr $ \Nothing -> Nothing, Nothing)
    go (Just a) = ((arr $ \(Just x) -> (fn, x)) >>> app >>> arr Just, (
```

parEval2

```
parEval2 :: (ArrowParallel arr a b conf,
    ArrowParallel arr (Maybe a, Maybe c) (Maybe b, Maybe d) conf,
   ArrowApply arr) =>
   conf -> arr a b -> arr c d -> (arr (a, c) (b, d))
parEval2 conf f g =
   (arr $ \(a, c) -> (f_g, [(Just a, Nothing), (Nothing, Just c)])) >>>
   app >>>
   (arr $ \comb -> (fromJust (fst (comb !! 0)), fromJust (snd (comb !!
where
   f_g = parEvalN conf $ replicate 2 $ arrMaybe f *** arrMaybe g
```

parMap

```haskell
parMap :: (ArrowParallel arr a b conf, ArrowApply arr) =>
  conf -> (arr a b) -> (arr [a] [b])
parMap conf f =
  (arr $ \as -> (f, as)) >>>
  ( first $ arr repeat >>>
    arr (parEvalN conf)) >>>
  app
```

```haskell
parMapStream :: (ArrowParallel arr a b conf, ArrowChoice arr, ArrowApply a
  conf -> ChunkSize -> arr a b -> arr [a] [b]
parMapStream conf chunkSize f =
  (arr $ \as -> (f, as)) >>>
  ( first $ arr repeat >>>
     arr (parEvalNLazy conf chunkSize)) >>>
  app
```

```
farm :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
  ArrowChoice arr, ArrowApply arr) =>
  conf -> NumCores -> arr a b -> arr [a] [b]
farm conf numCores f =
  (arr $ \as -> (f, as)) >>>
  ( first $ arr mapArr >>> arr repeat >>>
    arr (parEvalN conf)) >>>
  (second $ arr (unshuffle numCores)) >>>
  app >>>
  arr shuffle
```

The definition of unshuffle is

```
unshuffle :: Int
  -> [a]
  -> [[a]]
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]
```

, while shuffle is defined as:

```
shuffle :: [[a]]
  -> [a]
shuffle = concat . transpose
```

(These were taken from Eden's source code. [2])

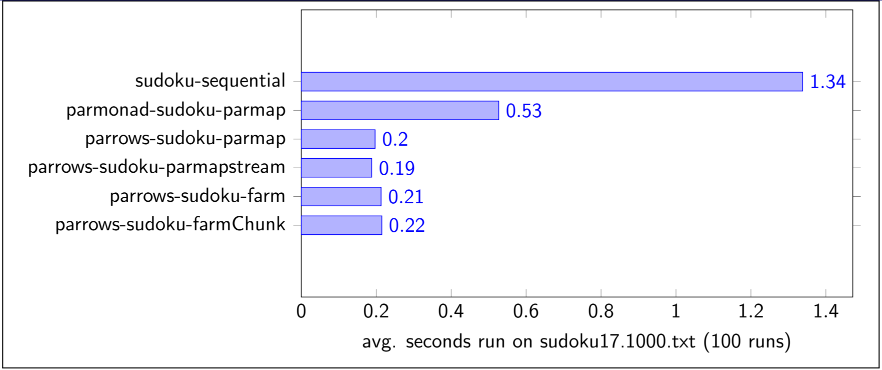
```
farmChunk :: (ArrowParallel arr a b conf, ArrowParallel arr [a] [b] conf,
  ArrowChoice arr, ArrowApply arr) =>
  conf -> ChunkSize -> NumCores -> arr a b -> arr [a] [b]
farmChunk conf chunkSize numCores f =
  (arr $ \as -> (f, as)) >>>
  (first $ arr mapArr >>> arr repeat >>>
    arr (parEvalNLazy conf chunkSize)) >>>
  (second $ arr (unshuffle numCores)) >>>
  app >>>
  arr shuffle
```

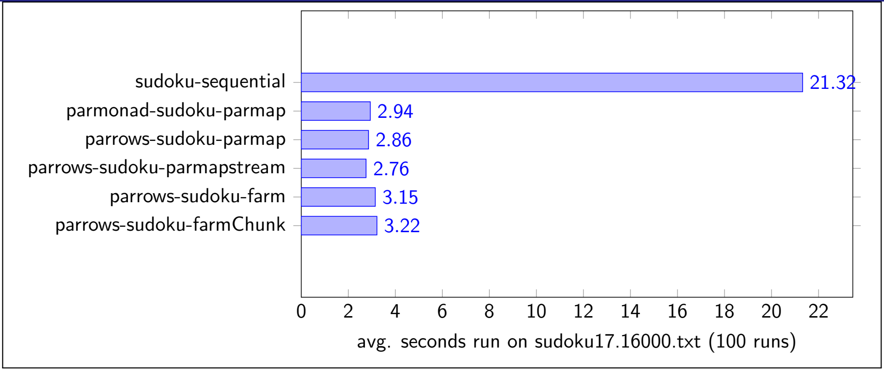
```haskell
(|>>>|) :: (Arrow arr) => [arr a b] -> [arr b c] -> [arr a c]
(|>>>|) = zipWith (>>>)
```

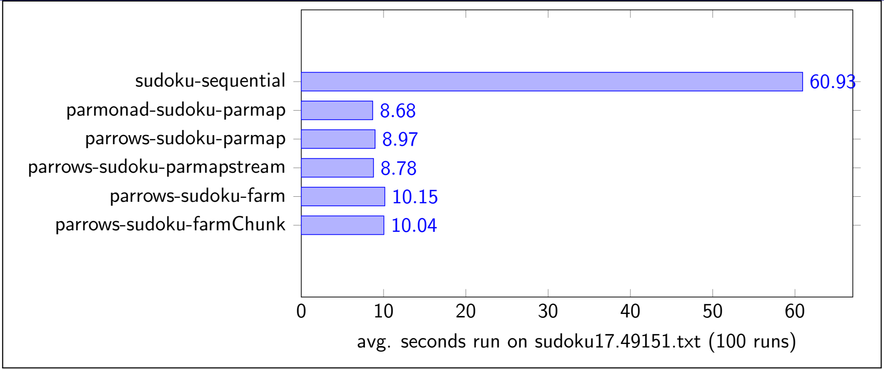
```haskell
(|***|) :: (ArrowParallel arr a b (),
   ArrowParallel arr (Maybe a, Maybe c) (Maybe b, Maybe d) (),
   ArrowApply arr) =>
   arr a b -> arr c d -> arr (a, c) (b, d)
(|***|) = parEval2 ()
```

```haskell
(|&&&|) :: (ArrowParallel arr a b (),
   ArrowParallel arr (Maybe a, Maybe a) (Maybe b, Maybe c) (),
   ArrowApply arr) =>
   arr a b -> arr a c -> arr a (b, c)
(|&&&|) f g = (arr $ \a -> (a, a)) >>> f |***| g
```

The Benchmarks were run on a Core i7-3970X CPU @ 3.5GHz with 6 cores and 12 threads. For sake of comparability with Simon Marlow's parallel version which uses the ParMonad, we use the ParMonad backend for the parallel arrow versions as well.


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
1.34
0.53
0.2
0.19
0.21
0.22
0
0.2
0.4
0.6
0.8
1
1.2
1.4
avg. seconds run on sudoku17.1000.txt (100 runs)


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
21.32
2.94
2.86
2.76
3.15
3.22
0
2
4
6
8
10
12
14
16
18
20
22
avg. seconds run on sudoku17.16000.txt (100 runs)


sudoku-sequential
parmonad-sudoku-parmap
parrows-sudoku-parmap
parrows-sudoku-parmapstream
parrows-sudoku-farm
parrows-sudoku-farmChunk
60.93
8.68
8.97
8.78
10.15
10.04
0
10
20
30
40
50
60
avg. seconds run on sudoku17.49151.txt (100 runs)

[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`.

[2] Eden skeletons' control.parallel.eden.map package source code. URL `https://hackage.haskell.org/package/edenskel-2.1.0.0/docs/src/Control-Parallel-Eden-Map.html`. [Accessed on 02/12/2017].