Functions

```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

Monad Definition

```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

Monad Usage

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 >>= ...
```

Arrow Definition

Another way to compose computations are arrows:

```
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 ∈ Arrows

Functions (->) are arrows:

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

The Kleisli Type

The Kleisli type

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

is also an arrow:

```haskell
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)
```

Combinators

Some Combinators:

```haskell
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)
```

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

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

Arrow Example

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)
```

1. Functional Programming 101
   - Short intro
   - Monads
   - Arrows

2. Parallel Arrows
   - Introduction to Parallelism
   - Generalization to Arrows
   - ArrowParallel Implementations

3. Usability
   - Skeletons
   - Syntactic Sugar

4. Benchmarks

In general, Parallelism can be looked at as:

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

Roadmap:

- Implement using existing Haskells
  - Multicore
  - ParMonad
  - Eden
- Generalize to Arrows
- Adapt Implementations
- Profit

Multicore Haskell

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

ParMonad

```haskell
parEvalN :: (NFData b) => [a -> b] -> [a] -> [b]
parEvalN fs as = runPar $
  (sequence $ map (spawnP) $ zipWith ($) fs as) >>= mapM get
```

with

```haskell
runPar :: Par a -> a
($) :: (a -> b) -> a -> b
sequence :: (Monad m) =>[m a] -> m [a]
map :: (a -> b) -> [a] -> [b]
spawnP :: NFData a =>a -> Par (IVar a)
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
(>>=) :: m a -> (a -> m b) -> m b
mapM :: Monad m =>(a -> m b) -> [a] -> m [b]
get :: IVar a -> Par a
```

Eden

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

with

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

The ArrowParallel typeclass

Now, let's generalize:

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

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

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

Multicore

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

with

```
listApp :: (ArrowChoice arr, ArrowApply arr) =>[arr a b] -> arr [a] [b]
(>>>) :: arr a b -> arr b c -> arr a c
arr :: Arrow arr =>(a -> b) -> arr a b
flip :: (a -> b -> c) -> b -> a -> c
using :: a -> Strategy a -> a
($) :: (a -> b) -> a -> b
parList :: Strategy a -> Strategy [a]
rdeepseq :: NFData a =>Strategy a
```

ParMonad

```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 sequence >>>
      arr (>>= mapM get) >>>
      arr runPar
```

with

```haskell
arr :: Arrow arr =>(a -> b) -> arr a b
zipWithArr :: ArrowChoice arr =>arr (a, b) c -> arr ([a], [b]) [c]
app :: ArrowApply arr =>(arr a b, a) b
spawnP :: NFData a =>a -> Par (IVar a)
sequence :: (Monad m) =>[m a] -> m [a]
(>>=) :: m a -> (a -> m b) -> m b
```

Eden (1)

For Eden we need separate implementations, for Functions:

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

with

```haskell
spawnF :: (Trans a, Trans b) =>[a -> b] -> [a] -> [b]
```

Eden (2)

and the Kleisli type:

```
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)
```

with

```
arr :: (Arrow arr) =>(a -> b) -> arr a b
map :: (a -> b) -> [a] -> [b]
sequence :: (Monad m) =>[m a] -> m [a]
```

Eden (3)

This is because of spawnF's signature:

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

and app's signature:

```
app :: (ArrowApply arr) => arr (arr a b, a) b
```

Eden (3)

This is because of spawnF's signature:

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

and app's signature:

```
app :: (ArrowApply arr) => arr (arr a b, a) b
```

Hacky alternative:

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

Skeletons... (1)

parEvalN, but **chunky**:

```
parEvalNLazy :: conf -> ChunkSize -> [arr a b] -> (arr [a] [b])
```

parallel evaluation of **different typed functions**:

```
parEval2 :: conf -> arr a b -> arr c d -> (arr (a, c) (b, d))
```

Skeletons... (2)

map, but in parallel:

```
parMap :: conf -> (arr a b) -> (arr [a]  [b])
```

parMap, but **chunky**:

```
parMapStream :: conf -> ChunkSize -> arr a b -> arr [a] [b]
```

Skeletons... (3)

parMap, but with **workload distribution**:

```
farm :: conf -> NumCores -> arr a b -> arr [a] [b]
```

farm, but **chunky**:

```
farmChunk ::
  conf -> ChunkSize -> NumCores -> arr a b-> arr [a] [b]
```

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

```
(|***|) :: (Arrow arr, ...) =>
  arr a b -> arr c d -> arr (a, c) (b, d)
(|***|) = parEval2 ()
```

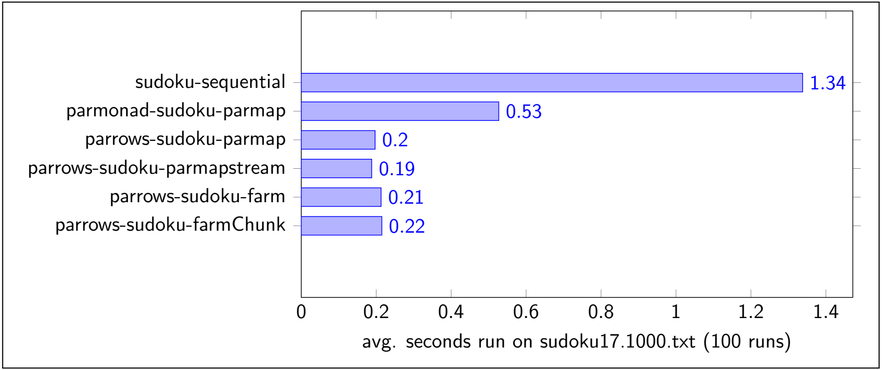
```
(|&&&|) :: (Arrow arr, ...) =>
  arr a b -> arr a c -> arr a (b, c)
(|&&&|) f g = (arr $ \a -> (a, a)) >>> f |***| g
```

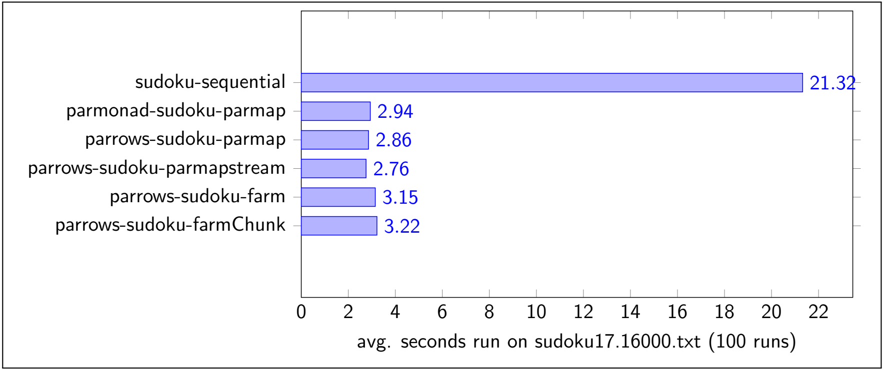
Parallelism as an operator

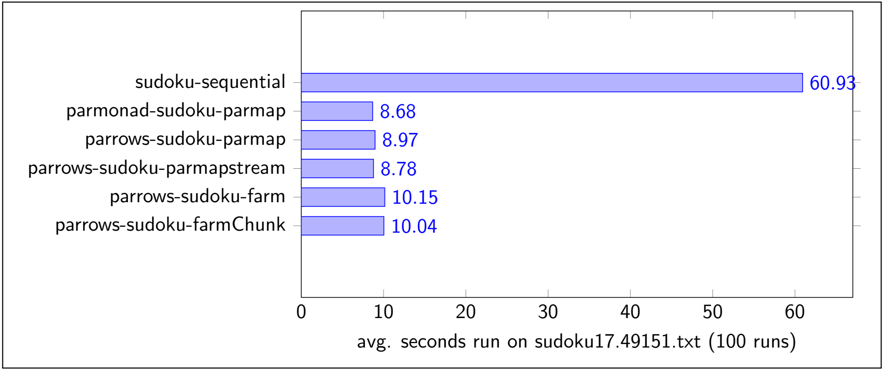
Parallel Evaluation made easy:

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

- Run on: Core i7-3970X CPU @ 3.5GHz / 6C/12T.
- compiled with ParMonad backend
- used Sudoku Benchmark from ParMonad examples


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)

Simon Marlow. Sample code for "Parallel and Concurrent Programming in Haskell". URL https://github.com/simonmar/parconc-examples. [Accessed on 01/12/2017].

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.

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].