Functions

```
1 public static int fib(int x) {
2   if (x<=0)
3     return 0;
4   else if (x==1)
5     return 1;
6   else
7     return fib(x-2) + fib(x-1);
8 }
```

```
1 fib :: Int -> Int
2 fib x
3   | x <= 0 = 0
4   | x == 1 = 0
5   | otherwise =
6     (fib (x - 2))
7       + (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

```
1 class Monad m where
2   (>>=) :: m a -> (a -> m b) -> m b
3   return :: a -> m a
```

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

```
1 instance Monad Maybe where
2   (Just a) >>= f = f a
3   Nothing >>= _ = Nothing
4   return a = Just a
```

$\Rightarrow$ composable computation descriptions

Monad Usage

With monadic functions like

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

we can compose computations:

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

Arrow Definition (1)

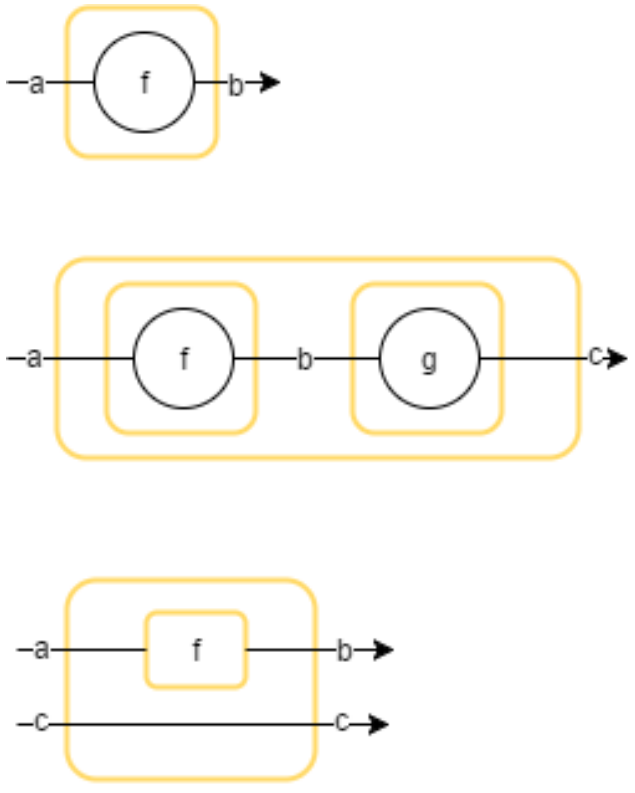
Another way to compose computations are arrows:

Arrow Definition (2)

```
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:

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

The Kleisli Type

The Kleisli type

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

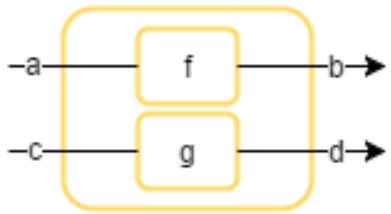
is also an arrow:

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

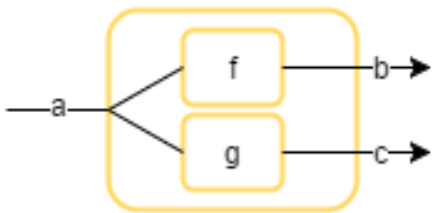
Combinators

```
1 second :: arr a b -> arr (c, a) (c, b)
2 second f = arr swap >>>
3     first f >>> arr swap
4   where swap (x, y) = (y, x)
```


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



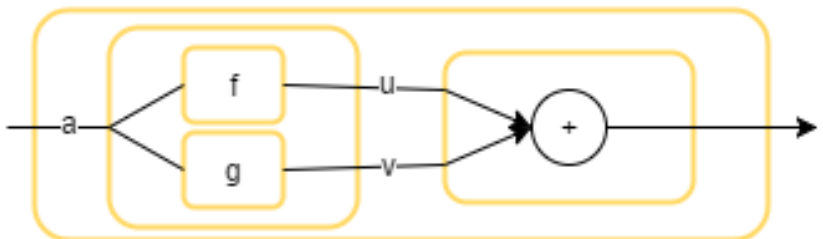

```
1 (&&&) :: arr a b -> arr a c ->
2     arr a (b, c)
3 f &&& g = arr (\a -> (a, a)) >>>
4     (f *** g)
```



Arrow Example

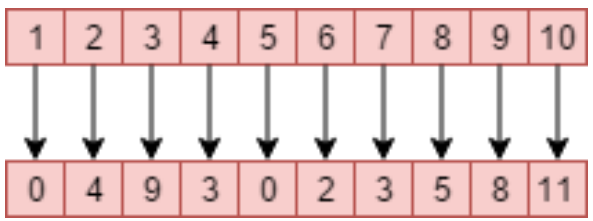
Arrow usage example:

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



In general, Parallelism can be looked at as:

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



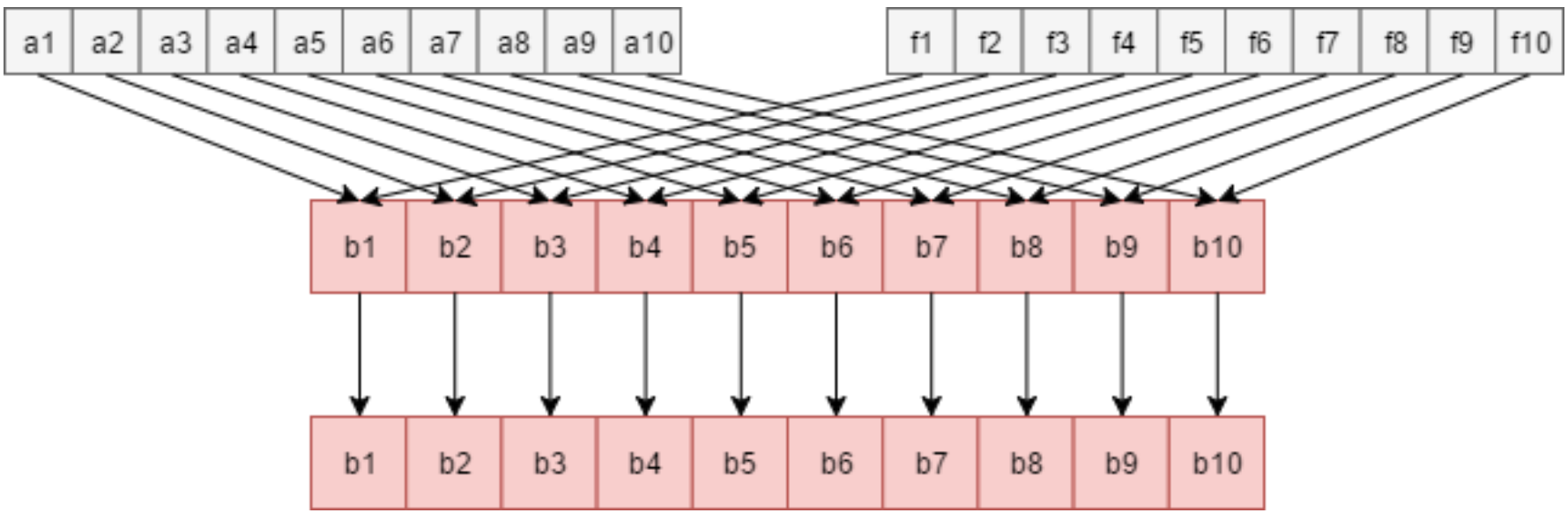
```
1 parEvalN :: [a -> b] -> [a] -> [b]
```

Roadmap:

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

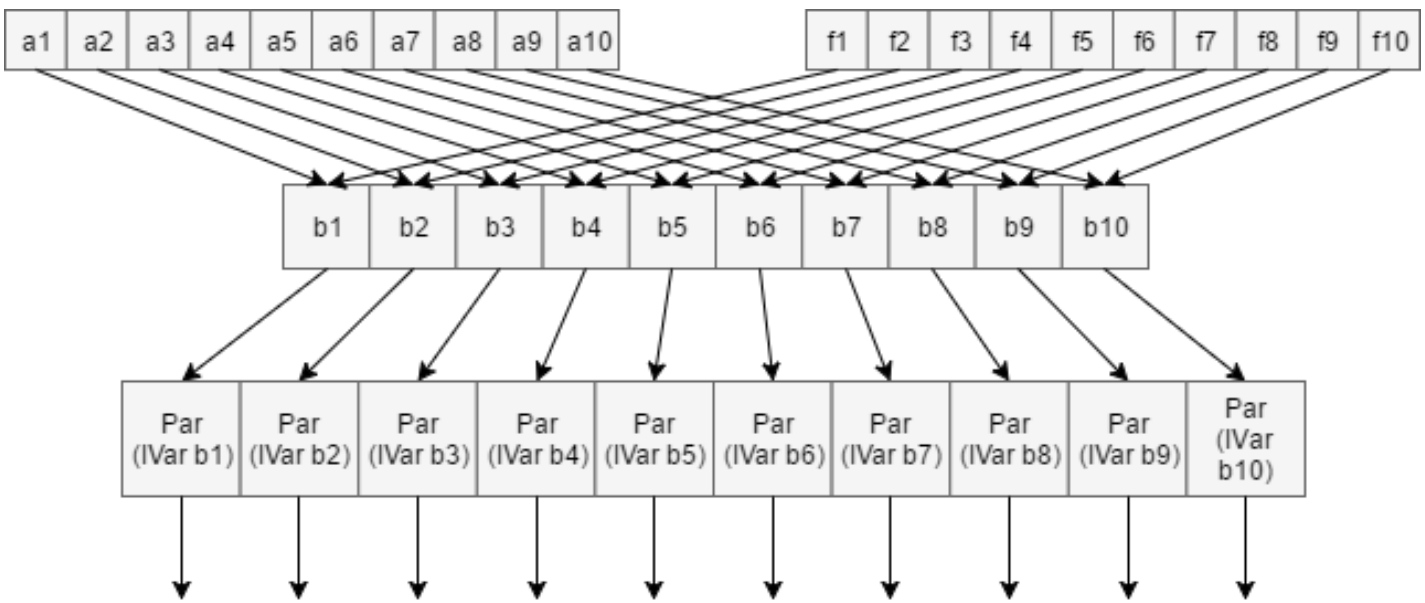
Multicore Haskell

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



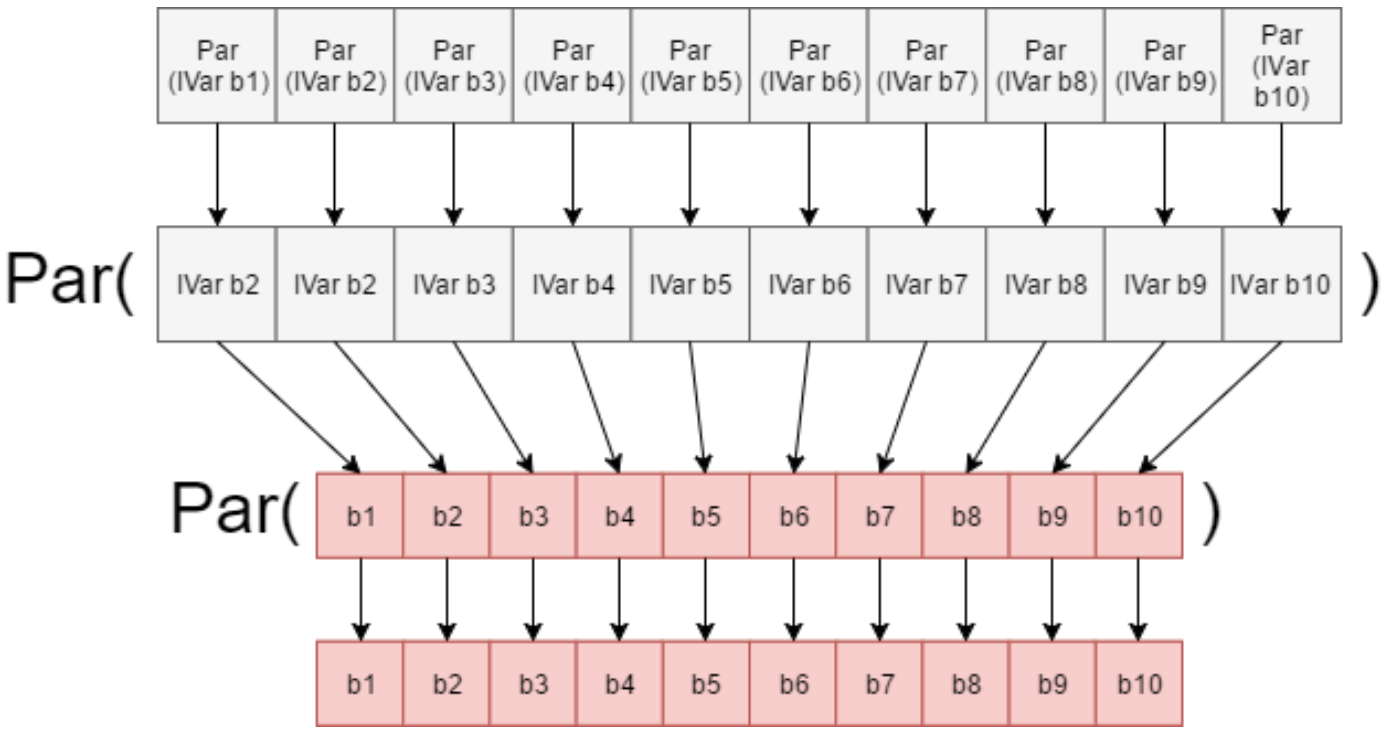
ParMonad

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

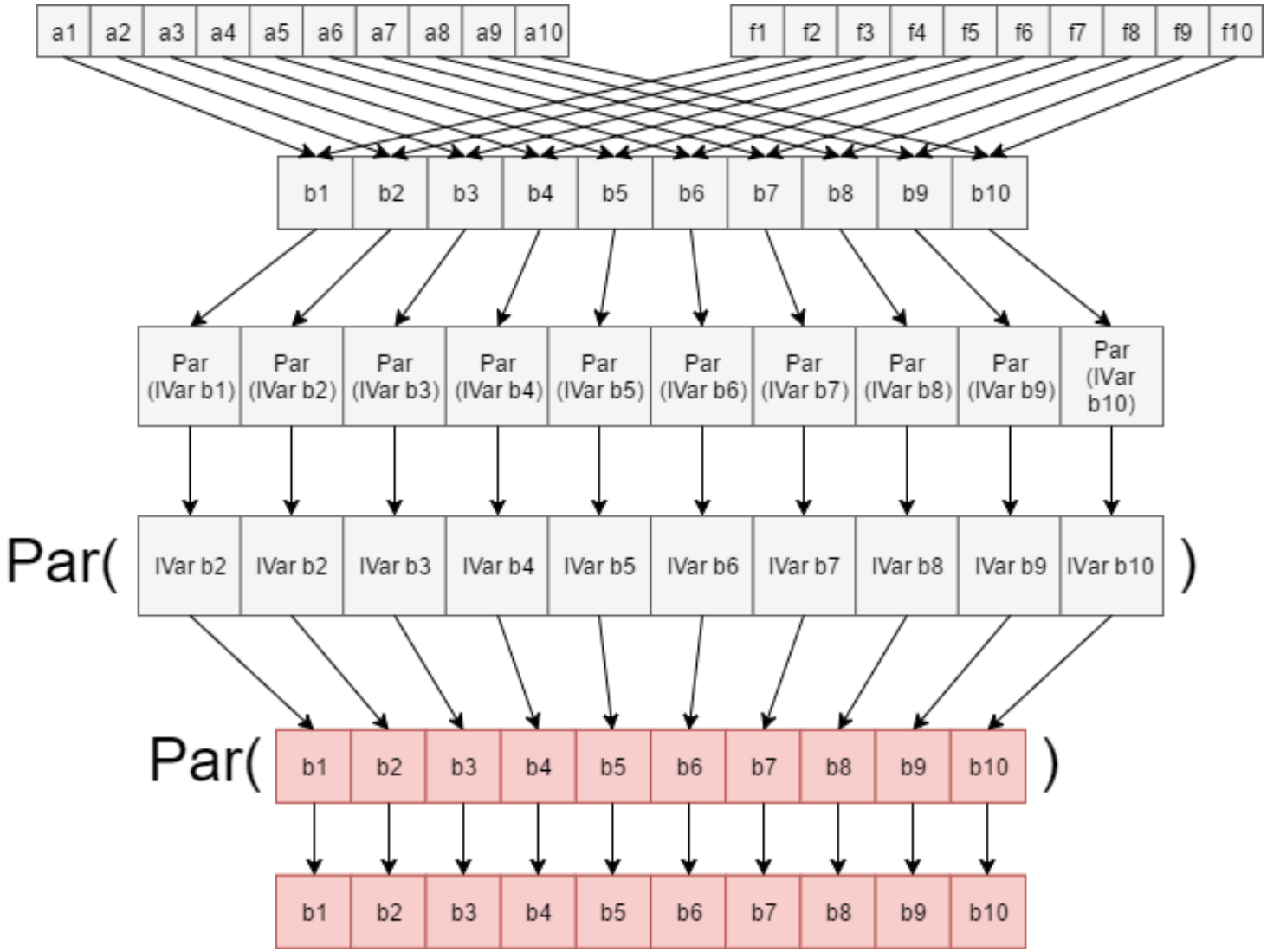


ParMonad

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

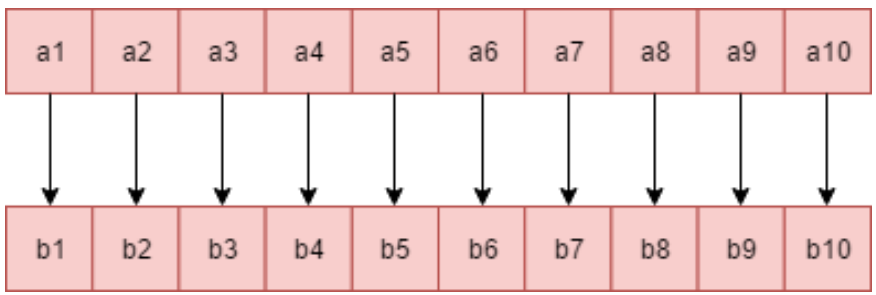


ParMonad


a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
f1 f2 f3 f4 f5 f6 f7 f8 f9 f10
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10
Par (IVar b1) Par (IVar b2) Par (IVar b3) Par (IVar b4) Par (IVar b5) Par (IVar b6) Par (IVar b7) Par (IVar b8) Par (IVar b9) Par (IVar b10)
Par(IVar b2 IVar b2 IVar b3 IVar b4 IVar b5 IVar b6 IVar b7 IVar b8 IVar b9 IVar b10)
Par(b1 b2 b3 b4 b5 b6 b7 b8 b9 b10)
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10

Eden

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



The ArrowParallel typeclass

Now, let's generalize:

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

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

The ArrowParallel typeclass

Now, let's generalize:

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

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

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

```
1 class Arrow arr => ArrowParallel arr a b conf where
2   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

```
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

```
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:

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

with

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:

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

and app's signature:

```
1 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
```

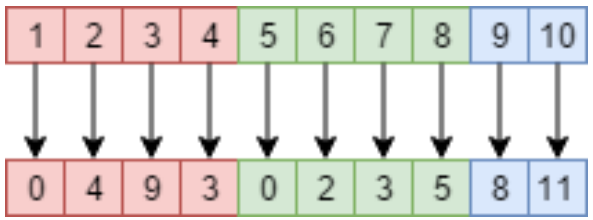
Hacky alternative:

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

Skeletons... (1)

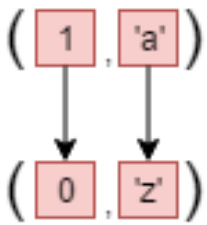
parEvalN, but **chunky**:

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



parallel evaluation of **different typed functions**:

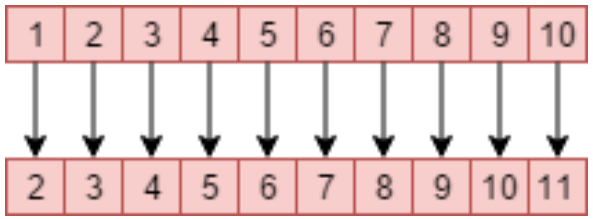
```
1 parEval2 :: conf -> arr a b -> arr c d -> (arr (a, c) (b, d))
```



Skeletons... (2)

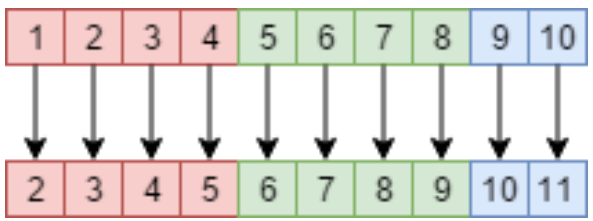
map, but in parallel:

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



parMap, but **chunky**:

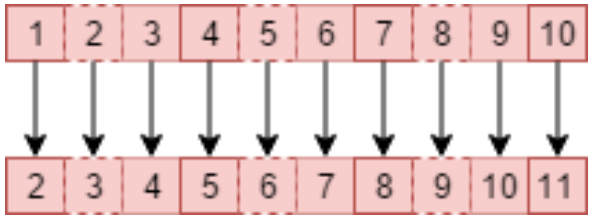
```
1 parMapStream :: conf -> ChunkSize -> arr a b -> arr [a] [b]
```



Skeletons... (3)

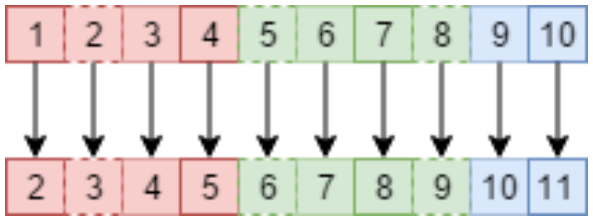
parMap, but with **workload distribution**:

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



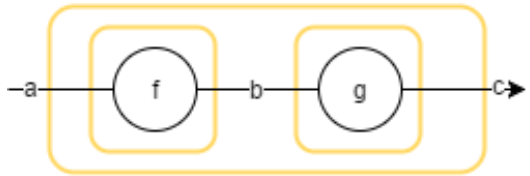
farm, but **chunky**:

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

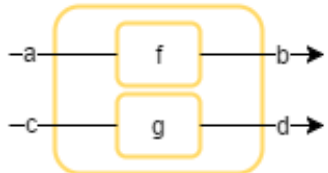


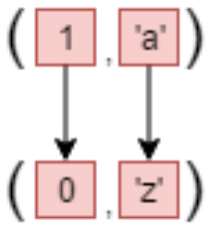
```
1 (|>>>|) :: (Arrow arr) => [arr a b] -> [arr b c] -> [arr a c]
2 (|>>>|) = zipWith (>>>)
```

On all Elements:

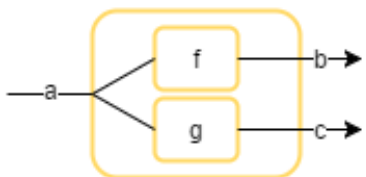


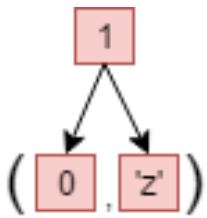

```
1 (|***|) :: (Arrow arr, ...) =>
2   arr a b -> arr c d -> arr (a, c) (b, d)
3 (|***|) = parEval2 ()
```





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

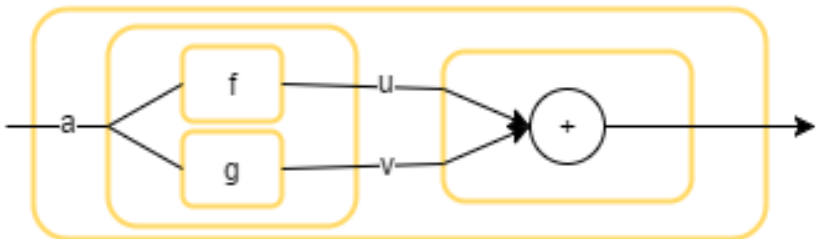


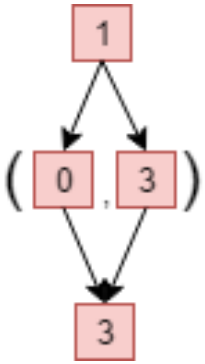


Parallelism as an operator

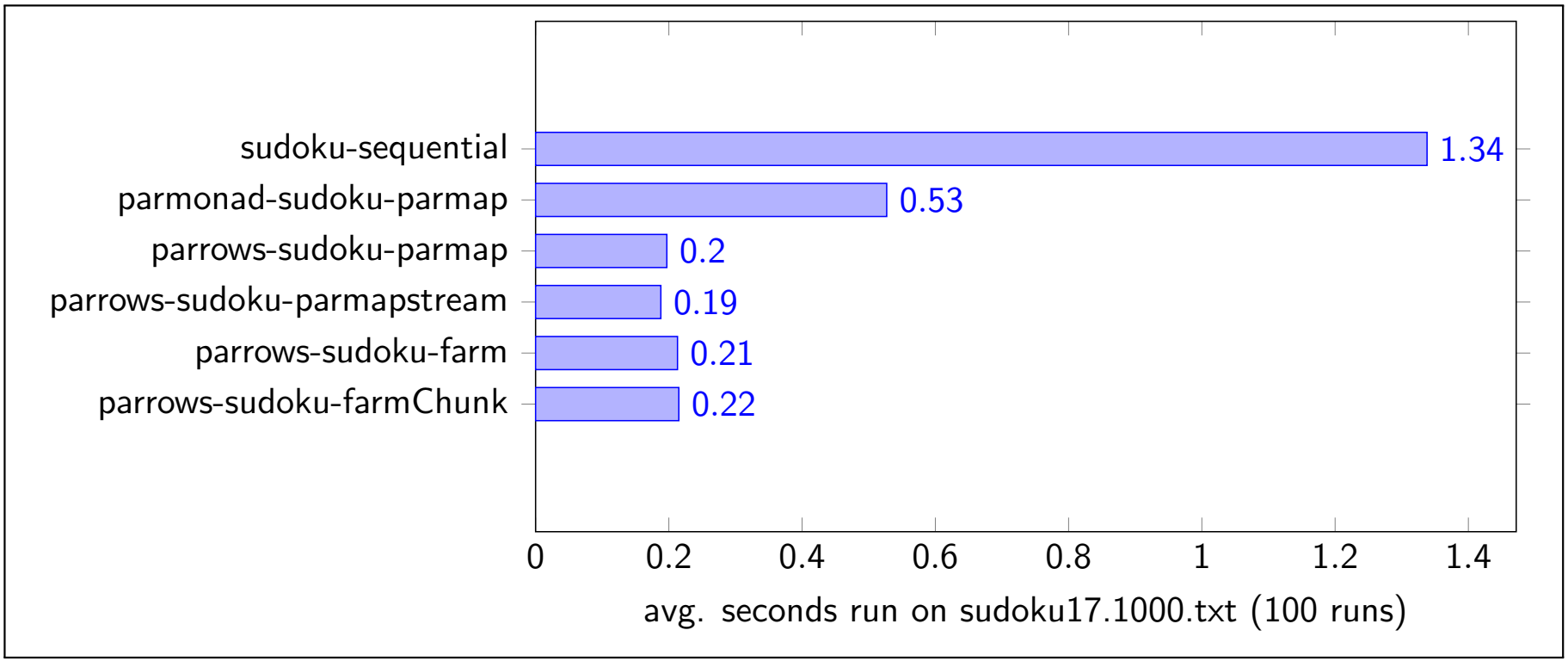
Parallel Evaluation made easy:

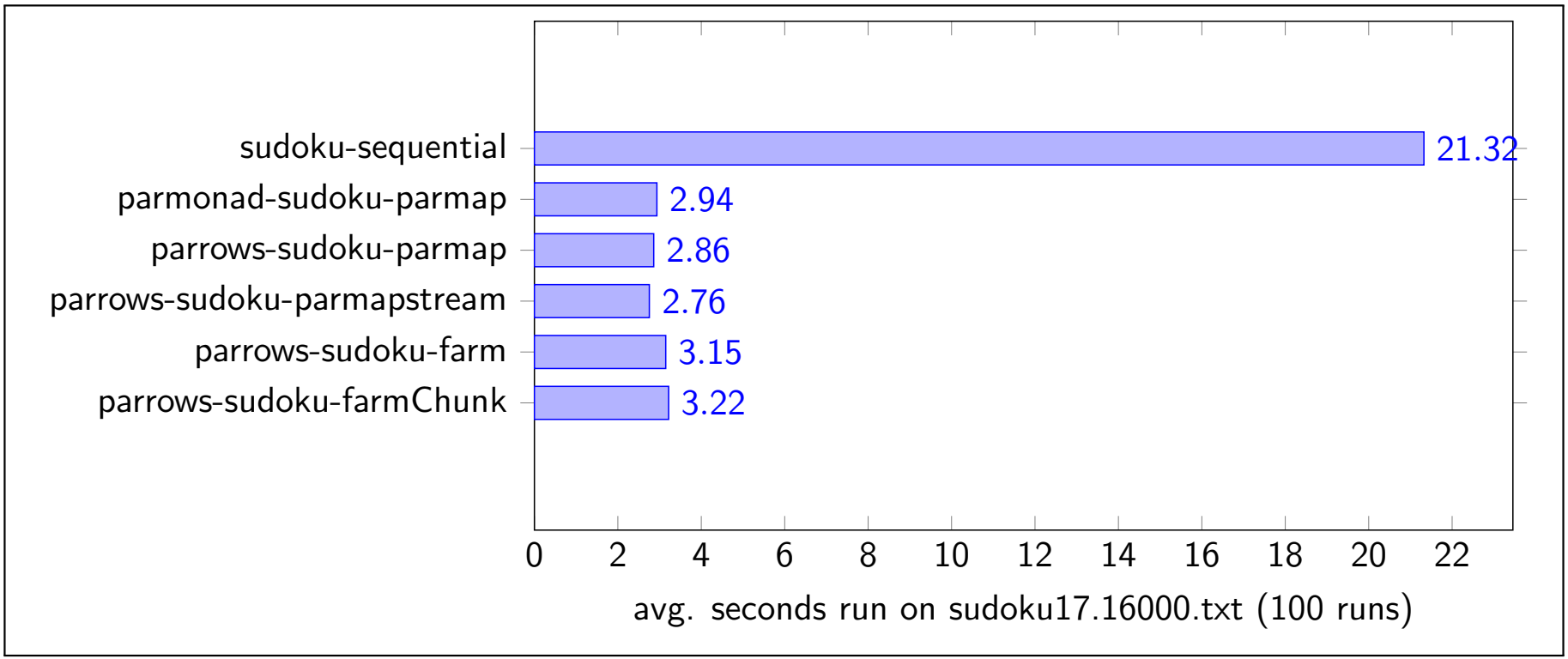
```
1 add :: Arrow arr => arr a Int -> arr a Int -> arr a Int
2 add f g = (f |&&&| g) >>> arr (\(u, v) -> u + v)
```

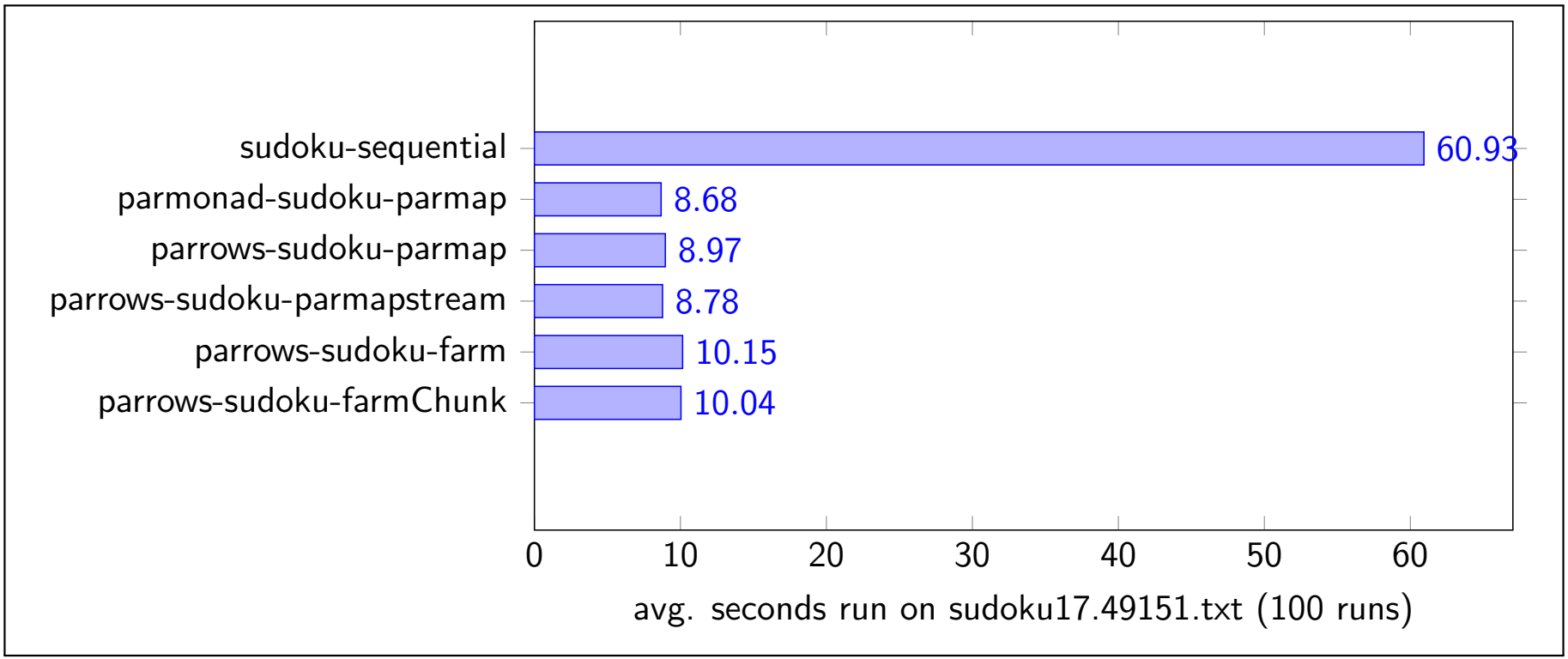




- Run on: Core i7-3970X CPU @ 3.5GHz / 6C/12T.
- compiled with ParMonad backend
- used Sudoku Benchmark from ParMonad examples


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)