

Arrows for Parallel Computation

Martin Braun

University of Bayreuth
martinbraun123@aol.com

January 24, 2018

1 Arrows 101

2 Parallel Arrows

- Introduction to Parallelism
- Generalization to Arrows
- ArrowParallel Implementations

3 Usability

- Syntactic Sugar
- Map Skeletons
- Futures
- Topology Skeletons

4 Further Notes

1 Arrows 101

2 Parallel Arrows

- Introduction to Parallelism
- Generalization to Arrows
- ArrowParallel Implementations

3 Usability

- Syntactic Sugar
- Map Skeletons
- Futures
- Topology Skeletons

4 Further Notes

Arrow Definition (1)

Another way to think about computations:

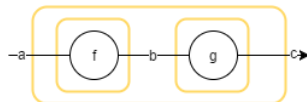
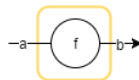


Arrow Definition (2)

```
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 \in Arrows

Functions (\rightarrow) are arrows:

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

The Kleisli Type

The Kleisli type

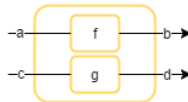
```
1 data Kleisli m a b = Kleisli { run :: a -> m b }
```

is also an arrow:

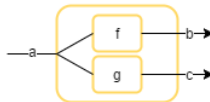
```
1 instance Monad m => Arrow (Kleisli m) where
2   arr f = Kleisli $ return . f
3   f >>> g = Kleisli $ \a -> f a >>= g
4   first f = Kleisli $ \(a,c) -> f a >>= \b -> return (b,c)
```

Combinators

- 1 $(***) :: \text{arr } a \ b \rightarrow \text{arr } c \ d \rightarrow \text{arr } (a, c) \ (b, d)$
- 2 $f *** g = \text{first } f >>> \text{second } g$



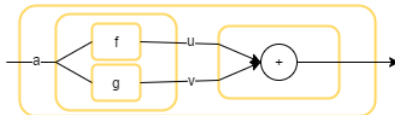
- 1 $(\&\&\&) :: \text{arr } a \ b \rightarrow \text{arr } a \ c \rightarrow \text{arr } a \ (b, c)$
- 2 $f \&\&\& g = \text{arr } (\backslash a \rightarrow (a, a)) >>> (f *** g)$



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)
```



1 Arrows 101

2 Parallel Arrows

- Introduction to Parallelism
- Generalization to Arrows
- ArrowParallel Implementations

3 Usability

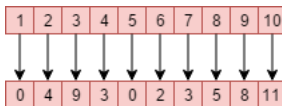
- Syntactic Sugar
- Map Skeletons
- Futures
- Topology Skeletons

4 Further Notes

Basic Parallelism (1)

In general, Parallelism can be looked at as:

1 $\text{parEvalN} :: [a \rightarrow b] \rightarrow [a] \rightarrow [b]$



Basic Parallelism (2)

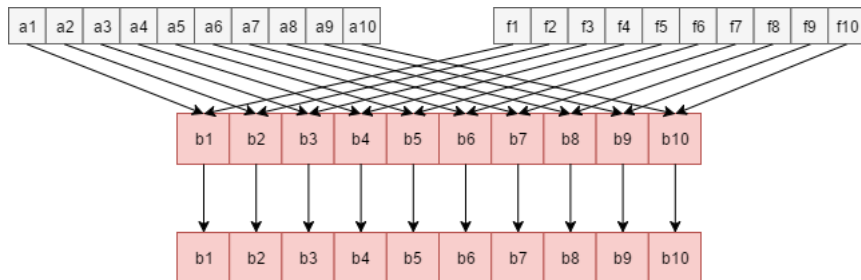
```
1 parEvalN :: [a -> b] -> [a] -> [b]
```

Roadmap:

- Implement using existing Haskell
 - GpH
 - ParMonad
 - Eden
- Generalize to Arrows
- Profit

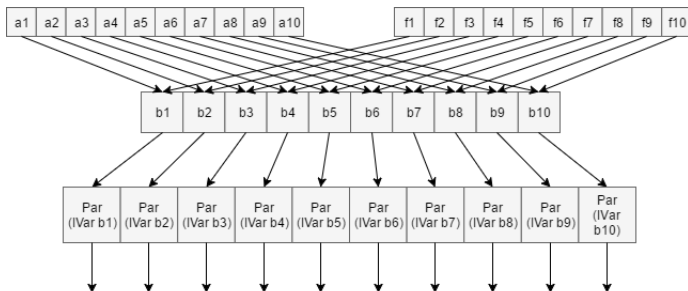
GpH

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



Par Monad

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

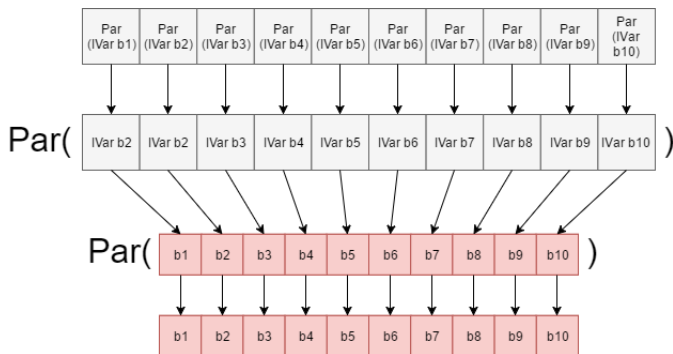


Par Monad

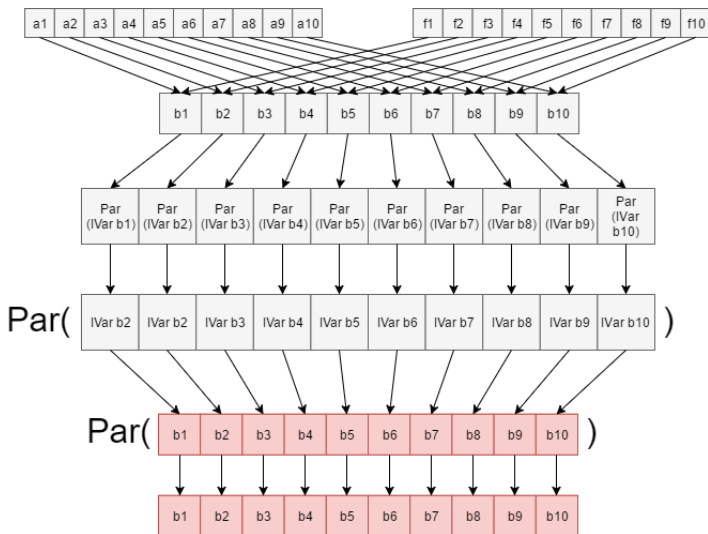
```

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

```

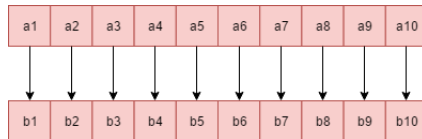


Par Monad



Eden

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



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]
```

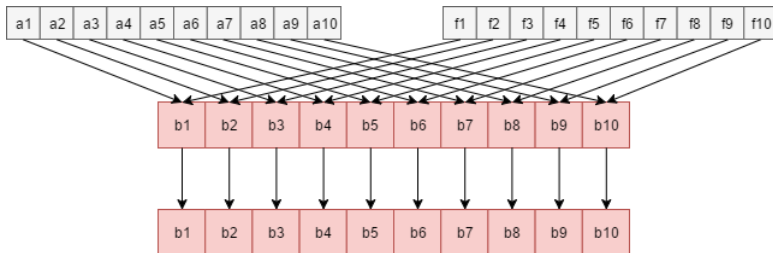
```
1 parEvalN :: (Arrow arr) => [arr a b] -> arr [a] [b]
```

```
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]
```

GpH

```
1 data Conf a = Conf (Strategy a)
2
3 instance (ArrowChoice arr) =>
4   ArrowParallel arr a b (Conf b) where
5     parEvalN (Conf strat) fs =
6       evalN fs >>>
7       arr (withStrategy (parList strat))
```

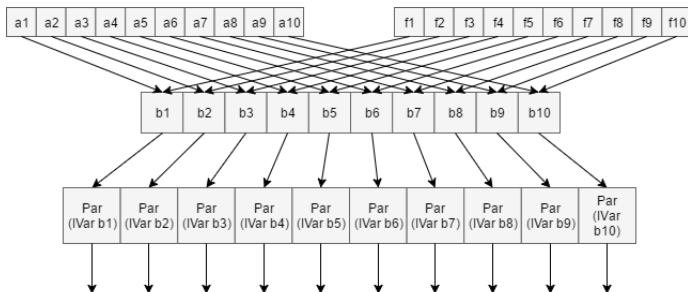


ParMonad

```

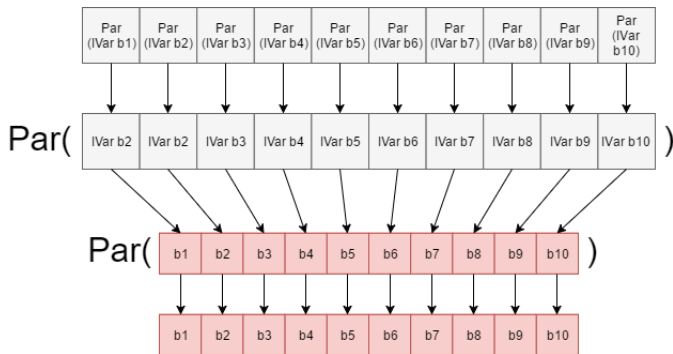
1 type Strategy a = a -> Par (IVar a)
2 data Conf a = Conf (Strategy a)
3
4 instance (ArrowChoice arr) => ArrowParallel arr a b (Conf b) where
5     parEvalN (Conf strat) fs =
6         evalN (map (>>> arr strat) fs) >>>
7         ...

```



ParMonad

```
1      ...  
2      arr sequenceA >>>  
3      arr (>>= mapM Control.Monad.Par.get) >>>  
4      arr runPar
```



Eden problems

For Eden we need separate implementations.

This is because of `spawnF` only supporting functions (\rightarrow) :

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

Eden problems

For Eden we need separate implementations.

This is because of `spawnF` only supporting functions `(->)`:

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

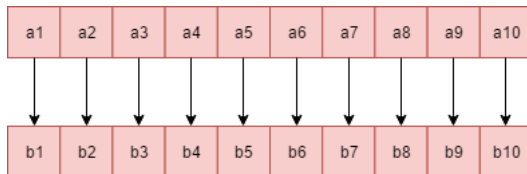
Hacky alternative:

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

Eden implementation - Functions

Straightforward:

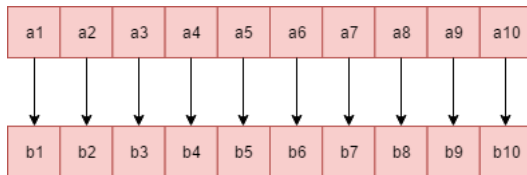
```
1 data Conf = Nil
2
3 instance (Trans a, Trans b) => ArrowParallel (->) a b conf where
4   parEvalN _ = spawnF
```



Eden implementation - Kleisli

Implementation for the Kleisli Type:

```
1 instance (ArrowParallel ( $\rightarrow$ ) a (m b) Conf,  
2   Monad m, Trans a, Trans b, Trans (m b)) =>  
3   ArrowParallel (Kleisli m) a b conf where  
4     parEvalN conf fs =  
5       arr (parEvalN conf (map (\(Kleisli f)  $\rightarrow$  f) fs)) >>>  
6       Kleisli sequence
```



1 Arrows 101

2 Parallel Arrows

- Introduction to Parallelism
- Generalization to Arrows
- ArrowParallel Implementations

3 Usability

- Syntactic Sugar
- Map Skeletons
- Futures
- Topology Skeletons

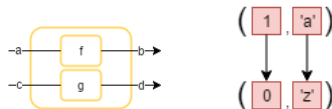
4 Further Notes

Parallel Operators

```

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

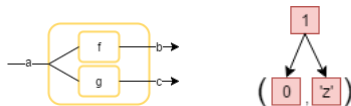
```



```

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

```



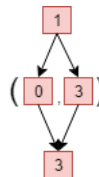
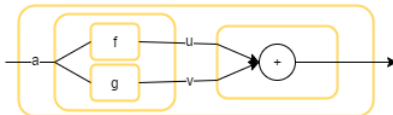
Parallelism made easy

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

```



Map Skeletons (1)

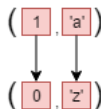
`parEvalN`, but **chunky**:

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



parallel evaluation of **different typed functions**:

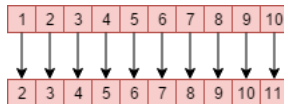
```
1 parEval2 :: conf -> arr a b -> arr c d -> arr (a, c) (b, d)
```



Map Skeletons (2)

map, but in **parallel**:

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



parMap, but **chunky**:

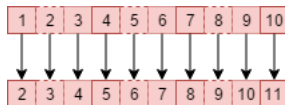
```
1 parMapStream :: conf -> ChunkSize -> arr a b -> arr [a] [b]
```



Map 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]
```



without Futures

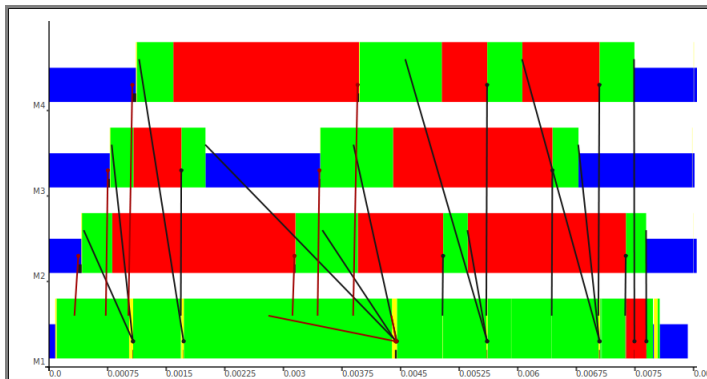
```
1 someCombinator :: [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 =
3   parEvalN () fs1 >>> rightRotate >>> parEvalN () fs2
```

without Futures

```

1 someCombinator :: [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 =
3   parEvalN () fs1 >>> rightRotate >>> parEvalN () fs2

```



Future definition

Since the particular concepts and implementations differ from backend to backend, we define the Future typeclass:

```
1 class Future fut a conf | a conf -> fut where  
2   put :: (Arrow arr) => conf -> arr a (fut a)  
3   get :: (Arrow arr) => conf -> arr (fut a) a
```

Future implementation (Eden)

```
1 data RemoteData a = RD { rd :: RD a }
2
3 put' :: (Arrow arr) => arr a (BasicFuture a)
4 put' = arr BF
5
6 get' :: (Arrow arr) => arr (BasicFuture a) a
7 get' = arr (\(~(BF a)) -> a)
8
9 instance NFData (RemoteData a) where
10     rnf = rnf . rd
11 instance Trans (RemoteData a)
12
13 instance (Trans a) => Future RemoteData a Conf where
14     put _ = put'
15     get _ = get'
```

with Futures

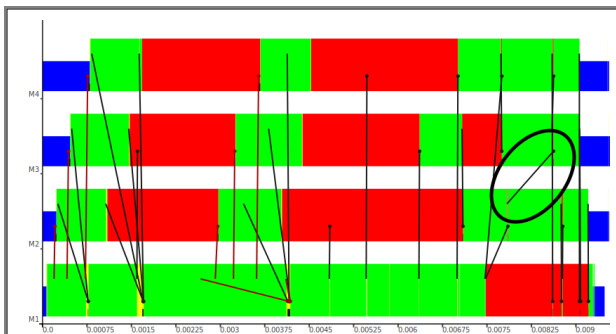
```
1 someCombinator :: [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 =
3   parEvalN () (map (>>> put ()) fs1) >>>
4   rightRotate >>>
5   parEvalN () (map (get ()) >>>) fs2)
```

with Futures

```

1 someCombinator :: [arr a b] -> [arr b c] -> arr [a] [c]
2 someCombinator fs1 fs2 =
3   parEvalN () (map (>>> put ()) fs1) >>>
4   rightRotate >>>
5   parEvalN () (map (get () >>>) fs2)

```



Simple Pipe

A simple Pipe (without Futures):

```
1 pipeSimple :: conf -> [arr a a] -> arr a a
2 pipeSimple conf fs =
3   loop (arr snd &&&
4     (arr (uncurry (:)) >>> lazy) >>> loopParEvalN conf fs)) >>>
5   arr last
```

Pipe with Futures

With Futures we get inter-node communication:

```
1 pipe :: conf -> [arr a a] -> arr a a
2 pipe conf fs =
3   unliftFut conf (pipeSimple conf (map (liftFut conf) fs))
```

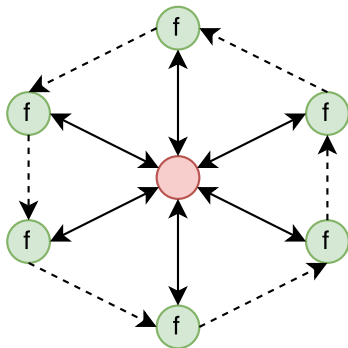
We can even implement

```
1 pipe2 :: conf -> arr a b -> arr b c -> arr a c
```

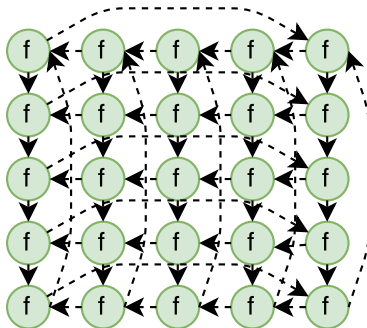
Which gives us parallel composition:

```
1 (|>>>|) :: arr a b -> arr b c -> arr a c
2 (|>>>|) = pipe2 ()
```

Ring



Torus



Profit

So... What does this get us?

- Arrow based Haskell \Rightarrow **Free Parallelism** for (other) Arrows
- **Replaceable Backends** \Rightarrow Easier Development
- Arrows are quite **intuitive** for parallelism

Further information

Paper submission:

<https://arxiv.org/pdf/1801.02216.pdf>

GitHub repository:

<https://github.com/s4ke/Parrows>

Frege Version in the works:

<https://goo.gl/oHbqh0>



Functional Programming 101

```
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?"
⇒ more concise ⇒ easier to reason about
- based on Lambda Calculus