Arrows for Parallel Computation

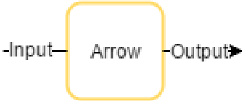
Martin Braun

University of Bayreuth
martinbraun123@aol.com

January 24, 2018

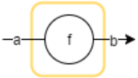
Arrow Definition (1)

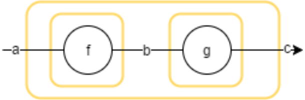
Another way to think about computations:

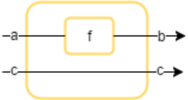


Arrow Definition (2)

```
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 ∈ Arrows

Functions (->) are arrows:

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

The Kleisli Type

The Kleisli type

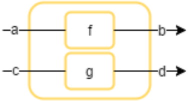
```
data Kleisli m a b = Kleisli { run :: a -> m b }
```

is also an arrow:

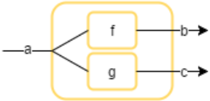
```
instance Monad m => Arrow (Kleisli m) where
  arr f = Kleisli $ return . f
  f >>> g = Kleisli $ \a -> f a >>= g
   first f = Kleisli $ \(a,c) -> f a >>= \b -> return (b,c)
```

Combinators

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



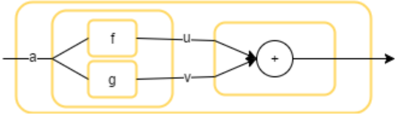

```
(&&&) :: arr a b -> arr a c -> arr a (b, c)
f &&& g = arr (\a -> (a, a)) >>> (f *** g)
```



Arrow Example

Arrow usage example:

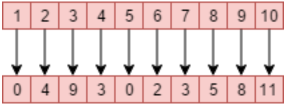
```
add :: Arrow arr => arr a Int -> arr a Int -> arr a Int
add f g = (f &&& g) >>> arr (\(u, v) -> u + v)
```



Basic Parallelism (1)

In general, Parallelism can be looked at as:

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



Basic Parallelism (2)

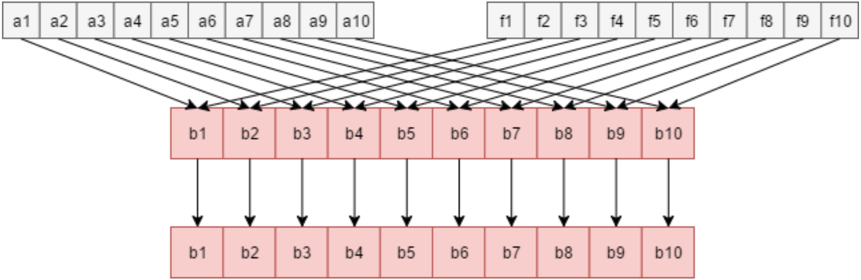
```
parEvalN :: [a -> b] -> [a] -> [b]
```

Roadmap:

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

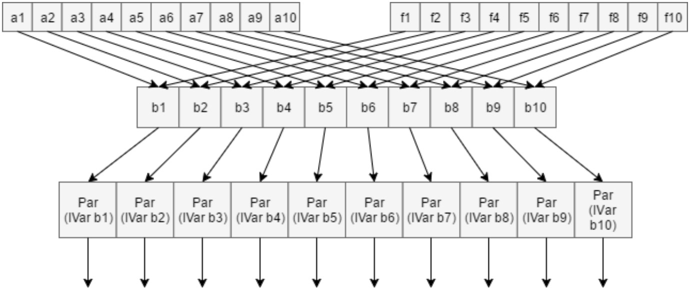
GpH

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



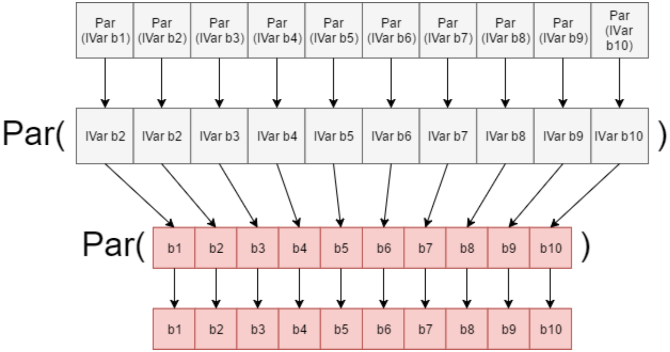
Par Monad

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

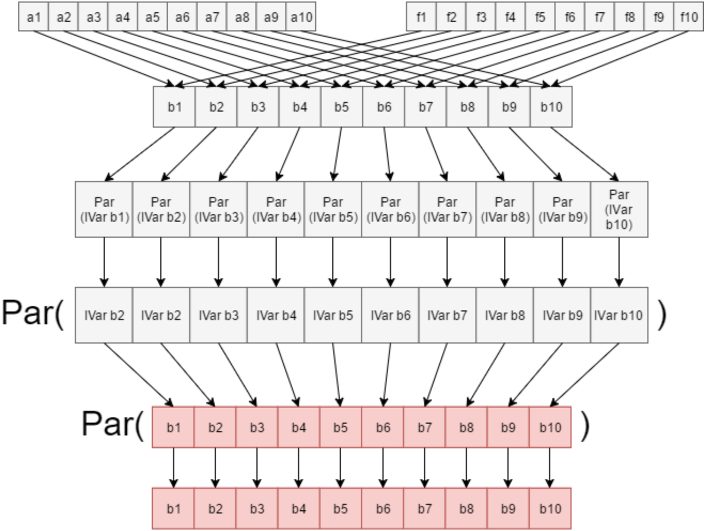


Par Monad

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

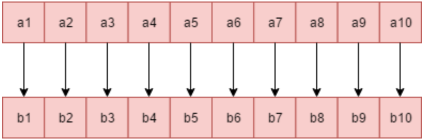


Par Monad


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

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



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]
```

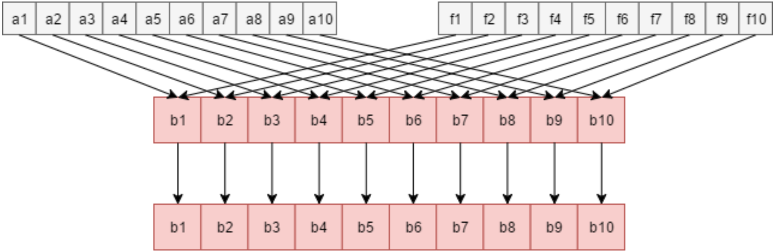
```
parEvalN :: (Arrow arr) => [arr a b] -> arr [a] [b]
```

```
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]
```

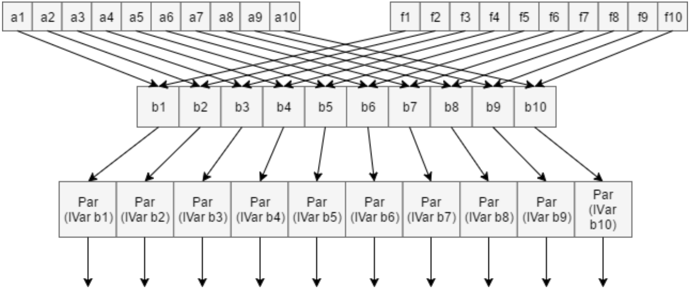
GpH

```haskell
data Conf a = Conf (Strategy a)

instance (ArrowChoice arr) =>
  ArrowParallel arr a b (Conf b) where
    parEvalN (Conf strat) fs =
        evalN fs >>>
        arr (withStrategy (parList strat))
```



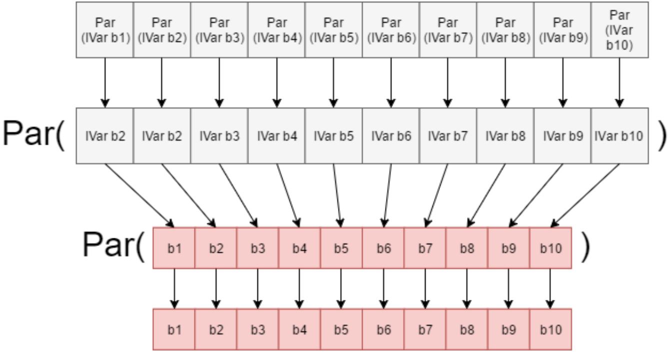
Par Monad

```haskell
type Strategy a = a -> Par (IVar a)
data Conf a = Conf (Strategy a)

instance (ArrowChoice arr) => ArrowParallel arr a b (Conf b) where
    parEvalN (Conf strat) fs =
        evalN (map (>>> arr strat) fs) >>>
         ...
```



ParMonad

```
...
arr sequenceA >>>
arr (>>= mapM Control.Monad.Par.get) >>>
arr runPar
```



Eden problems

For Eden we need separate implementations.

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

```
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 (->):

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

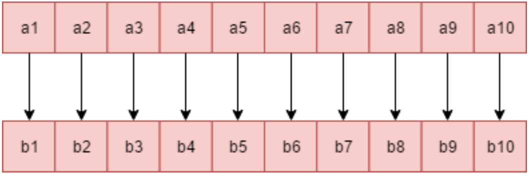
Hacky alternative:

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

Eden implementation - Functions

Straightforward:

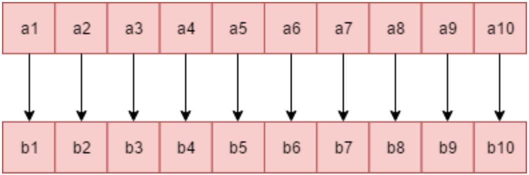
```
data Conf = Nil

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



Eden implementation - Kleisli

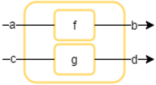
Implementation for the Kleisli Type:

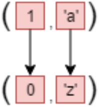
```
instance (ArrowParallel (->) a (m b) Conf,
  Monad m, Trans a, Trans b, Trans (m b)) =>
   ArrowParallel (Kleisli m) a b conf where
    parEvalN conf fs =
       arr (parEvalN conf (map (\(Kleisli f) -> f) fs)) >>>
        Kleisli sequence
```



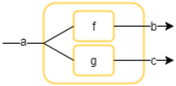
Parallel Operators

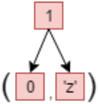
```haskell
(|***|) :: arr a b -> arr c d -> arr (a, c) (b, d)
(|***|) = parEval2 ()
```






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

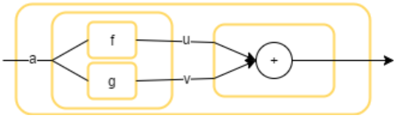


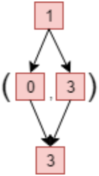


Parallelism made easy

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)
```

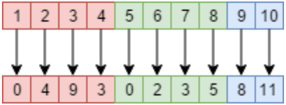




Map Skeletons (1)

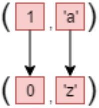
parEvalN, but **chunky**:

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



parallel evaluation of **different typed functions**:

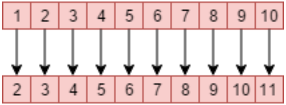
```
parEval2 :: conf -> arr a b -> arr c d -> arr (a, c) (b, d)
```



Map Skeletons (2)

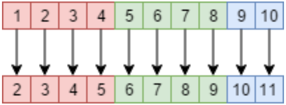
map, but in **parallel**:

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



parMap, but **chunky**:

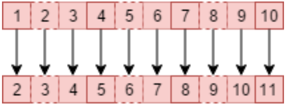
```
parMapStream :: conf -> ChunkSize -> arr a b -> arr [a] [b]
```



Map Skeletons (3)

parMap, but with **workload distribution**:

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



farm, but **chunky**:

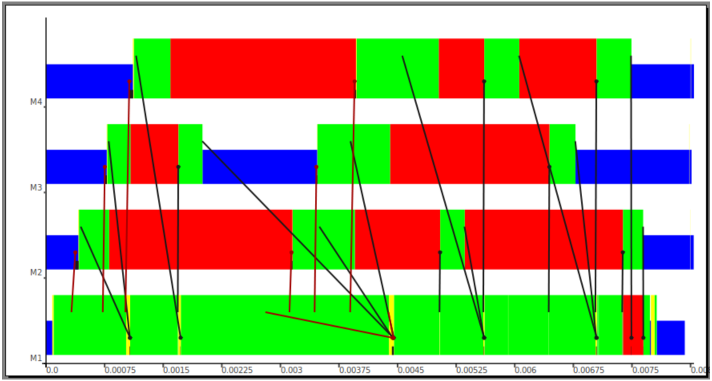
```
farmChunk ::
  conf -> ChunkSize -> NumCores -> arr a b-> arr [a] [b]
```

without Futures

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

without Futures

```
someCombinator :: [arr  a  b]  −> [arr b c]  −> arr [a]  [c]
someCombinator fs1 fs2 =
  parEvalN ()  fs1  >>> rightRotate >>> parEvalN () fs2
```



Future definition

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

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

Future implementation (Eden)

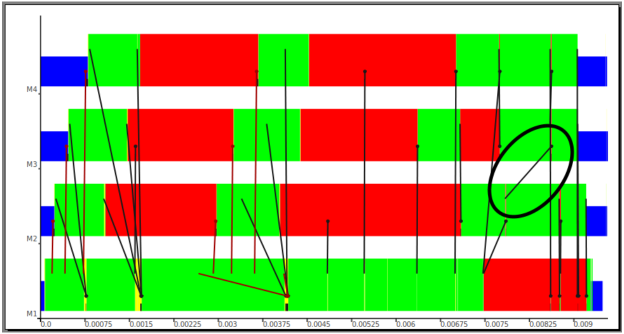
```
data RemoteData a = RD { rd :: RD a }

put' :: (Arrow arr) => arr a (BasicFuture a)
put' = arr BF

get' :: (Arrow arr) => arr (BasicFuture a) a
get' = arr (\(~(BF a)) -> a)

instance NFData (RemoteData a) where
    rnf = rnf . rd
instance Trans (RemoteData a)

instance (Trans a) => Future RemoteData a Conf where
    put _ = put'
    get _ = get'
```

with Futures

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

with Futures

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



Simple Pipe

A simple Pipe (without Futures):

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

Pipe with Futures

With Futures we get inter-node communication:

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

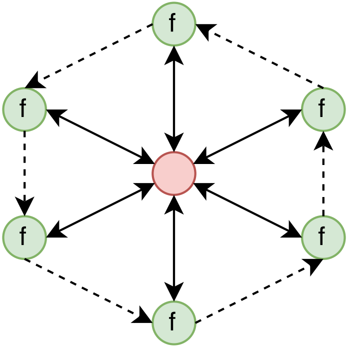
We can even implement

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

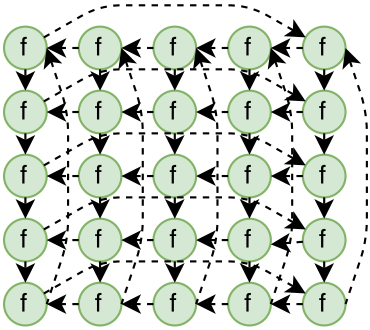
Which gives us parallel composition:

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

Ring



Torus



Profit

So... What does this get us?

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

Further information

Paper submission:
`https://arxiv.org/pdf/1801.02216.pdf`

GitHub repository:
`https://github.com/s4ke/Parrows`

Functional Programming 101

```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

pipe2

```
pipe2 :: conf -> arr a b -> arr b c -> arr a c
pipe2 conf f g =
    (arr return &&& arr (const [])) &&& arr (const []) >>>
    pipe conf (replicate 2 (unify f g)) >>>
    arr snd >>>
    arr head
    where
        unify :: arr a b -> arr b c ->
            arr (([a], [b]), [c]) (([a], [b]), [c])
        unify f' g' = (mapArr f' *** mapArr g') *** arr (const [])
            >>> arr (\((b, c), a) -> ((a, b), c))
```

ring

```haskell
ring :: conf -> arr (i, r) (o, r) -> arr [i] [o]
ring conf f =
    loop (second (rightRotate >>> lazy) >>>
        arr (uncurry zip) >>>
        loopParEvalN conf
            (repeat (second (get conf) >>> f >>>
                second (put conf))) >>>
        arr unzip) >>>
    postLoopParEvalN conf (repeat (arr id))
```

torus

```
torus :: conf -> arr (c, a, b) (d, a, b) -> arr [[c]] [[d]]
torus conf f =
    loop (second ((mapArr rightRotate >>> lazy)
            *** (arr rightRotate >>> lazy)) >>>
        arr (uncurry3 (zipWith3 lazyzip3)) >>>
        arr length &&& (shuffle >>>
            loopParEvalN conf (repeat (ptorus conf f))) >>>
        arr (uncurry unshuffle) >>>
        arr (map unzip3) >>> arr unzip3 >>> threetotwo) >>>
    postLoopParEvalN conf (repeat (arr id))
```