Contents

1 General Introduction **2**

2 Short introduction to parallel Haskells **2**
- 2.1 Multicore Haskell . . . 2
- 2.2 ParMonad . . . 2
- 2.3 Eden . . . 2
- 2.4 HdpH . . . 2

3 Arrows **2**

4 Utility Functions **3**

5 Parallel Arrows **4**
- 5.1 Multicore Haskell . . . 5
- 5.2 ParMonad . . . 5
- 5.3 Eden . . . 6
- 5.4 HdpH . . . 6

6 Skeletons **6**
- 6.1 parEvalNLazy . . . 6
- 6.2 parEval2 . . . 7
- 6.3 parMap . . . 7
- 6.4 parMapStream . . . 8
- 6.5 farm . . . 8
- 6.6 farmChunk . . . 9

7 Syntactic Sugar **9**

8 Benchmarks **10**

9 Conclusion **10**

1 General Introduction

Arrows were introduced in John Hughes paper as an alternative to Monads for API design.[1]In the paper Hughes describes that Arrows have one powerful additional property when compared to Monads for API design: Extensibility. In this paper we will show how this property can be used to add parallelism capabilities to different arrows.

2 Short introduction to parallel Haskells

2.1 Multicore Haskell

2.2 ParMonad

2.3 Eden

2.4 HdpH

3 Arrows

John Hughes defined the Arrow typeclass as follows[2]:

```
class Arrow a where
  arr  ::  (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
   first   ::  a b c -> a (b,d) (c,d)
```

`arr :: (b -> c) -> a b c` is used to lift an ordinary function to an Arrow type. This can be thought of as analogous to the monadic **return**. The >>> operator, in a similar way, is analogous to the monadic composition operator >>=. Ant lastly, the `first` operator, which takes the input arrow from b to c and converts it into an arrow on pairs with the second argument untouched, is also needed for actual useful code as without it, we wouldn't have a way to save input across arrows.

The most prominent instances of this interface are regular functions, (−>)

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

and the Kleisli type.

```
instance Arrow (Kleisli  m) where
  arr  f =  Kleisli  $ return . f
```

[1]John Hughes, Generalising Monads to Arrows, see [1]
[2]John Hughes, Generalising Monads to Arrows, see [1]

```
  f >>> g = Kleisli $ \b -> f b >>= g
   first  f =  Kleisli  $ \(b,d)  -> f b >>= \c -> return (c,d)
```

With this typeclass in place, Hughes also defined some syntactic sugar: The mirrored version of first , called second,

```
second :: Arrow a => a b c -> a (d, b) (d, c)
second f = arr swap >>> first f >>> arr swap
  where swap (x, y) = (y, x)
```

the *** combinator which combines first and second to handle two inputs in one arrow,

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

and the &&& combinator that constructs an arrow which outputs 2 different values like ***, but takes only one input.

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

A short example given by Hughes on how to use this is add over arrows:

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

The benefit of this interface is now that any type which is shown to be an Arrow can no be used in conjunction with this newly created add combinator. Even though this example is quite simple, the power of the Arrow interface immediately is clear: If a type is an Arrow, it can immediately used together with every library that works on Arrows. With simple Monads this type of extensibility is not possible.

Note: In the definitions above we used the notation a b c *for an arrow from* b *to* c*. From now on we will use the equivalent defintion* arr a b *for an arrow from* a *to* b *to make it easier to find the arrow type in type signatures. We kept the original notation* a b c *for this section to not change too much from Hughes' original definitions.*

4 Utility Functions

Before we go into detail on parallel arrows, we introduce some utility combinators first, that will help us later: map and zipWith on arrows.
The map combinator lifts any arrow arr a b to an arrow arr [a] [b],

```
-- from http://www.cse.chalmers.se/~rjmh/afp-arrows.pdf
mapArr :: ArrowChoice arr => arr a b -> arr [a] [b]
mapArr f =
  arr listcase >>>
```

```haskell
  arr (const []) ||| (f *** mapArr f >>> arr (uncurry (:)))
  where
    listcase [] = Left ()
    listcase (x:xs) = Right (x,xs)
```

and zipWith lift any arrow `arr (a, b) c` to an arrow `arr ([a], [b]) [c]`.

```haskell
zipWithArr :: ArrowChoice arr => arr (a, b) c -> arr ([a], [b]) [c]
zipWithArr f = (arr $ \(as, bs) -> zipWith (,) as bs) >>> mapArr f
```

These two combinators make use of the `ArrowChoice` typeclass, which allows us to use the `|||` combinator, which takes two arrows `arr a c` and `arr b c` and combines them into a new arrow `arr (Either a b) c` which pipes all **Left** `a`'s to the first arrow and all **Right** `b`'s to the second arrow.

```haskell
 (|||) :: ArrowChoice arr a c -> arr b c -> arr (Either a b) c
```

With the zipWithArr combinator we can also write a combinator `listApp`, which lifts a list of arrows `[arr a b]` to an arrow `arr [a] [b]`.

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

This combinator also makes use of the ArrowApply typeclass which allows us to evaluate arrows with `app :: arr (arr a b, a) c`.

Do we need the exact definition of ArrowChoice here?

5 Parallel Arrows

We have seen what Arrows are, so we can take a look at how they can be used to define a general interface not just to computation, but to **parallel computation** as well, next.

In its purest form, parallel computation (on functions) can be looked at as the execution of some functions `a -> b` in parallel:

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

Translating this into arrow terms gives us a new operator `parEvalN` that lifts a list of arrows `[arr a b]` to an parallel arrow `arr [a] [b]` (This combinator is similar to listApp from 4, which does no parallel evaluation of `[arr a b]`).

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

As the implementation may a) differ depending on the type of arrow and b) we want this to be an interface for different backends with, we introduce the new typeclass `ArrowParallel`.

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

As computation in some parallel Haskells involves additional configuration parameters giving info about the environment, we also introduce an additional conf parameter to the function which can be of different type for each backend, so we add it to the type signature of our class as well.

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

Note that we don't require the conf parameter in every implementation. If it is not needed, we allow the conf type parameter to be of any type and don't even evaluate it by blanking it in the type signature of the implemented parEvalN.

5.1 Multicore Haskell

The Multicore Haskell implementation of this class is straightforward using listApp from 4 combined with the using operator from Multicore Haskell.

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

We hardcode the parList rdeepseq strategy here as in this context it is the only one making sense as we usually want the output list to be fully evaluated to its normal form.

5.2 ParMonad

The ParMonad implementation makes use of Haskells laziness and ParMonad's spawnP :: a -> Par (IVar a) function which forks away the computation of a value and returns an IVar containing the result in the Par monad.

We therefore apply each function to its corresponding input value with app and then fork the computation away with arr spawnP inside a zipWithArr call. This yields a list [Par (IVar b)], which we then convert into Par [IVar b] with arr sequenceA. In order to wait for the computation to finish, we map over the IVars inside the ParMonad with arr (>>= mapM get). The result of this operation is a Par [b] from which we can finally remove the monad again by running arr runPar to get our output of [b].

```
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
```

5.3 Eden

For the Multicore and ParMonad implementation we could use general implementations that just required the ArrowApply and ArrowChoice interface. With Eden this is not the case as we can only spawn a list of functions and we cannot extract simple functions out of arrows. While this could be "fixed" by introducing a typeclass like

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

we don't do this in this paper, as this seems too hacky. For now, we just implement ArrowParallel for normal functions

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

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)
```

5.4 HdpH

6 Skeletons

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

6.1 parEvalNLazy

parEvalN is 100% strict, which means that it fully evaluates all passed arrows. Sometimes this might not be feasible, as it will not work on infinite lists of functions like e.g. map (arr . (+)) [1..] or just because we need the arrows evaluated in chunks. parEvalNLazy fixes this. It works by first chunking the input from [a] to [[a]] with the given ChunkSize in (arr $ chunksOf chunkSize). These chunks are then fed into a list [arr [a] [b]] of parallel arrows created by feeding chunks of the passed ChunkSize into the regular parEvalN by using listApp. The resulting [[b]] is lastly converted into [b] with arr concat.

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

```haskell
  where
    fchunks = map (parEvalN conf) $ chunksOf chunkSize fs
```

6.2 parEval2

We have only talked about parallelizing the computation of arrows of the same type up until now. But sometimes we want to paralellize inhomogenous types as well. For this, we introduce a helper combinator arrMaybe first, that converts an arrow arr a b to an arrow arr (Maybe a) (Maybe b).

```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, (Just a))
```

With this, we can now easily write parEval2 which combines two arrows arr a b and arr c d into a new parallel arrow arr (a, c) (b, d). We start by converting both arrows with arrMaybe, combining them with *** into a new arrow arr (Maybe a, Maybe c) (Maybe b, Maybe d). This is then replicated twice and fed into parEvalN to get a arr [(Maybe a, Maybe c)] [(Maybe b, Maybe d)]. We can then apply this arrow to the input [(Just a, Nothing), (Nothing, Just c)] and then extract the resulting values with **fromJust** and the !! operator on lists in the last step.

```haskell
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 !! 1))))
where
  f_g = parEvalN conf $ replicate 2 $ arrMaybe f *** arrMaybe g
```

6.3 parMap

parMap is probably the most common skeleton for parallel programs. We can implement it with ArrowParallel by repeating an arrow arr a b and then passing it into parEvalN to get an arrow arr [a] [b]. Just like parEvalN, parMap is 100 % strict.

```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
```

6.4 parMapStream

As parMap is 100% strict it has the same restrictions as parEvalN compared to parEvalNLazy. So it makes sense to also have a parMapStream which behaves like parMap, but uses parEvalNLazy instead of parEvalN.

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

6.5 farm

parMap spawns every single computation in a new thread (at least for the instances of ArrowParallel we gave in this paper). This can be quite wasteful and a farm that equally distributes the workload over numCores workers (if numCores ¿ actualProcessorCount, the fastest processor(s) to finish will get more tasks) seems useful.

```
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.

Do we want a farm that works on a list of arrows as well?

For this we just have to replace the mapArr with zipWithArr

6.6 farmChunk

As farm is basically just parMap with a different work distribution, it is, again, 100% strict. So we define farmChunk which uses parEvalNLazy instead of parEvalN like this:

```
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
```

7 Syntactic Sugar

To make using our new API a bit easier, we also introduce some syntactic sugar. We start with |>>>|, which is basically >>> on lists of arrows.

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

For the basic Arrow case, we have the *** combinator which allows us to combine two arrows arr a b and arr c d into an arrow arr (a, c) (b, d) which does both computations at once. This can easily be lifted into a parallel computation with parEval2, but requires a backend which has a implementation that does not require any configuration (hence the () as the conf parameter in the following code snippet).

```
(|***|) :: (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 ()
```

With this we can analogously to the serial &&& define the parallel |&&&|.

```
(|&&&|) :: (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
```

8 Benchmarks

needed for this version of the paper? we have to find better benchmarks anyways

9 Conclusion

References

[1] Generalising Monads to Arrows `https://jcp.org/en/jsr/detail?id=220`, November 10, 1998