

# Parallel Functional Programming

## Lecture 2

Mary Sheeran

(with thanks to Simon Marlow for use of slides)

<http://www.cse.chalmers.se/edu/course/pfp>

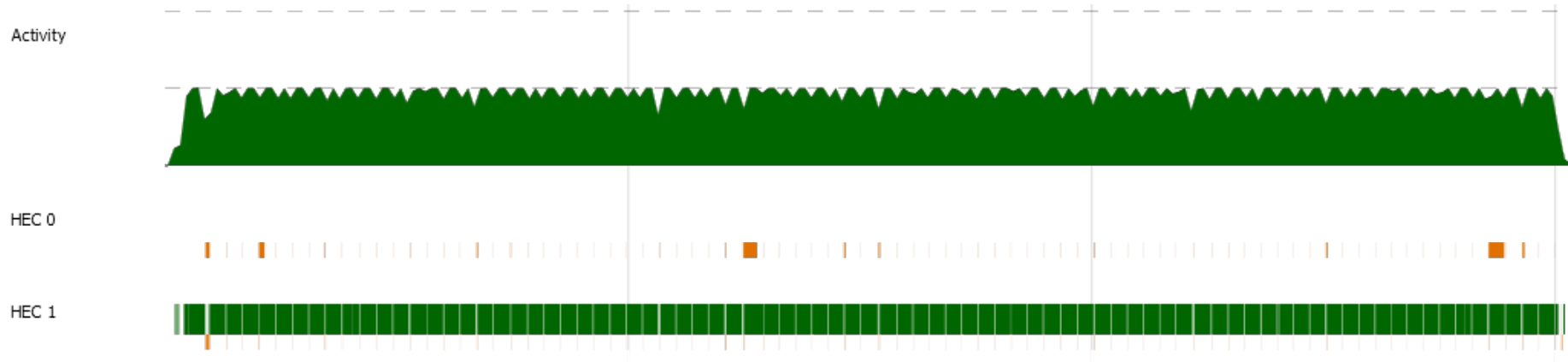
# Remember nfib

```
nfib :: Integer -> Integer  
nfib n | n < 2 = 1  
nfib n = nfib (n-1) + nfib (n-2) + 1
```

- A trivial function that returns the number of calls made—and makes a very large number!

n	nfib n
10	177
20	21891
25	242785
30	2692537

# Sequential



# Explicit Parallelism

`par x y`

- “Spark” `x` in parallel with computing `y`
  - (and return `y`)
- The run-time system *may* convert a spark into a parallel task—or it may not
- Starting a task is cheap, but not free

# Explicit Parallelism

`x `par` y`

# Explicit sequencing

`pseq x y`

- Evaluate  $x$  *before*  $y$  (and return  $y$ )
- Used to *ensure* we get the right evaluation order

# Explicit sequencing

`x `pseq` y`

- Binds more tightly than `par`

# Using par and pseq

```
import Control.Parallel

rfib :: Integer -> Integer
rfib n | n < 2 = 1
rfib n = nf1 `par` nf2 `pseq` nf2 + nf1 + 1
    where nf1 = rfib (n-1)
          nf2 = rfib (n-2)
```



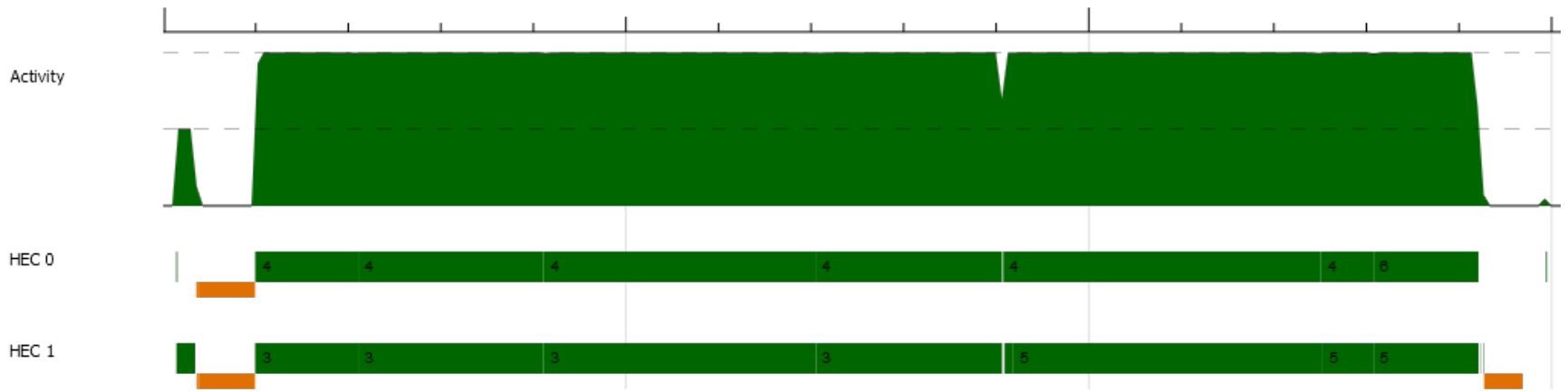
# Using par and pseq

```
import Control.Parallel

rfib :: Integer -> Integer
rfib n | n < 2 = 1
rfib n = nf1 `par` (nf2 `pseq` nf2 + nf1 + 1)
  where nf1 = rfib (n-1)
        nf2 = rfib (n-2)
```

- Evaluate *nf1 in parallel with* (Evaluate *nf2 before ...*)

# Looks promising

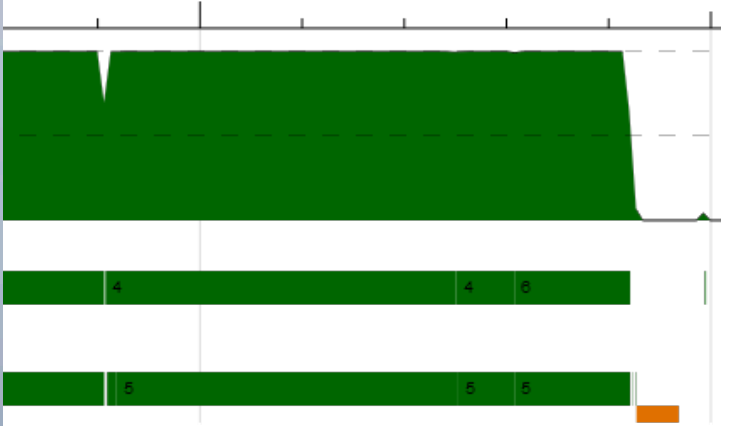


missing

Activity

HEC 0

HEC 1



but is way too slow

# What's up?

```
> Nfib +RTS -N2 -s
```

-s to get stats

# Hah

29860703

...

**SPARKS: 14963763 (29 converted, 10353439 pruned)**

INIT time 0.02s ( 0.00s elapsed)

MUT time 2.00s ( 0.98s elapsed)

GC time 0.19s ( 0.12s elapsed)

EXIT time 0.00s ( 0.00s elapsed)

Total time 2.20s ( 1.10s elapsed)

...

# Hah

29860703

...

**SPARKS: 14963763 (29 converted, 10353439 pruned)**

INIT time 0.02s ( 0.00s elapsed)

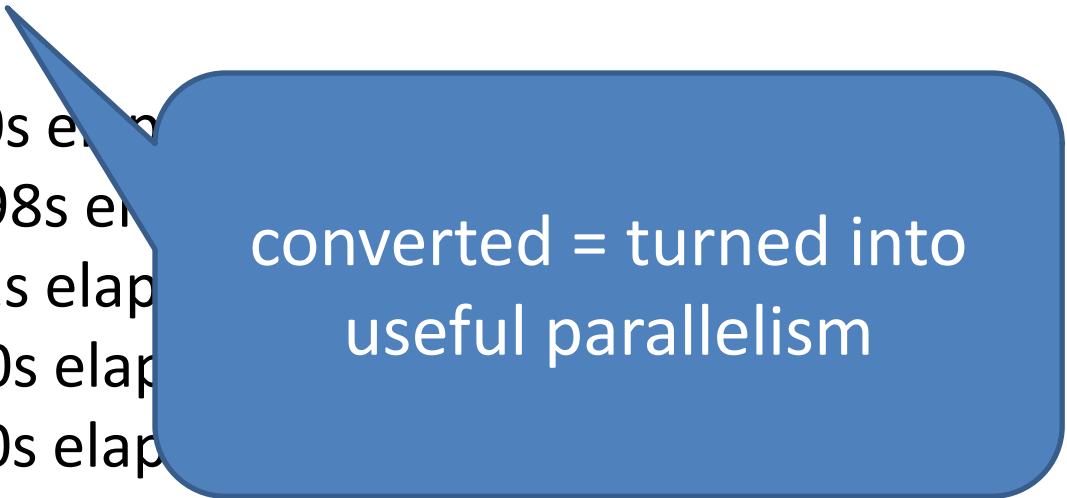
MUT time 2.00s ( 0.98s elapsed)

GC time 0.19s ( 0.12s elapsed)

EXIT time 0.00s ( 0.00s elapsed)

Total time 2.20s ( 1.10s elapsed)

...



converted = turned into  
useful parallelism

# Controlling Granularity

- Let's use a threshold for going sequential, **t**

```
tfib :: Integer -> Integer -> Integer
tfib t n | n < t = nfib n
tfib t n = nf1 `par` nf2 `pseq` nf1 + nf2 + 1
  where nf1 = tfib t (n-1)
        nf2 = tfib t (n-2)
```

# Better

tfib 30 35

gives

SPARKS: 20 (4 converted, 16 pruned)

INIT time	0.00s	( 0.00s elapsed)
MUT time	1.47s	( 0.81s elapsed)
GC time	0.00s	( 0.01s elapsed)
EXIT time	0.00s	( 0.00s elapsed)
Total time	1.47s	( 0.83s elapsed)



# Controlling evaluation degree

May want to force evaluation of some expressions (on left of par, pseq)

Remember John's question

# Too lazy evaluation?

This only evaluates the *first constructor* of the list!

```
psort [] = []  
psort (x:xs) = par rest $  
                psort [y | y <- xs, y < x]  
                ++ [x]  
                ++ rest  
where rest = psort [y | y <- xs, y >= x]
```

- What would happen if  
par (rnf rest)?

rnf means fully evaluate  
See also RWH ch. 24  
Are results still the same?

# What are we controlling?

The division of the work into possible parallel tasks (**par**) including choosing size of tasks

GHC runtime takes care of choosing which sparks to actually evaluate in parallel and of distribution

Need also to control order of evaluation (**pseq**) and degree of evaluation

**Dynamic behaviour** is the term used for how a pure function gets partitioned, distributed and run

Remember, this is deterministic parallelism. The answer is always the same!

# positive so far (par and pseq)

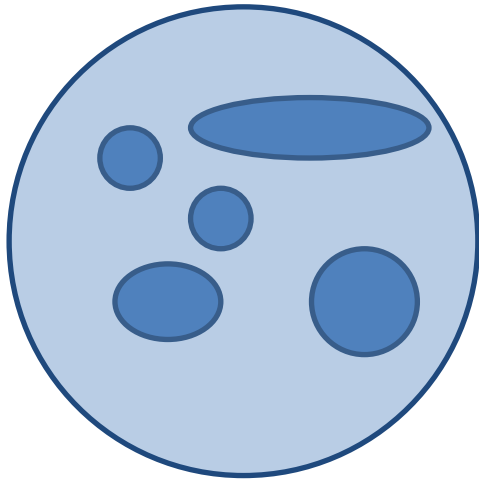
Don't need to

- express communication

- express synchronisation

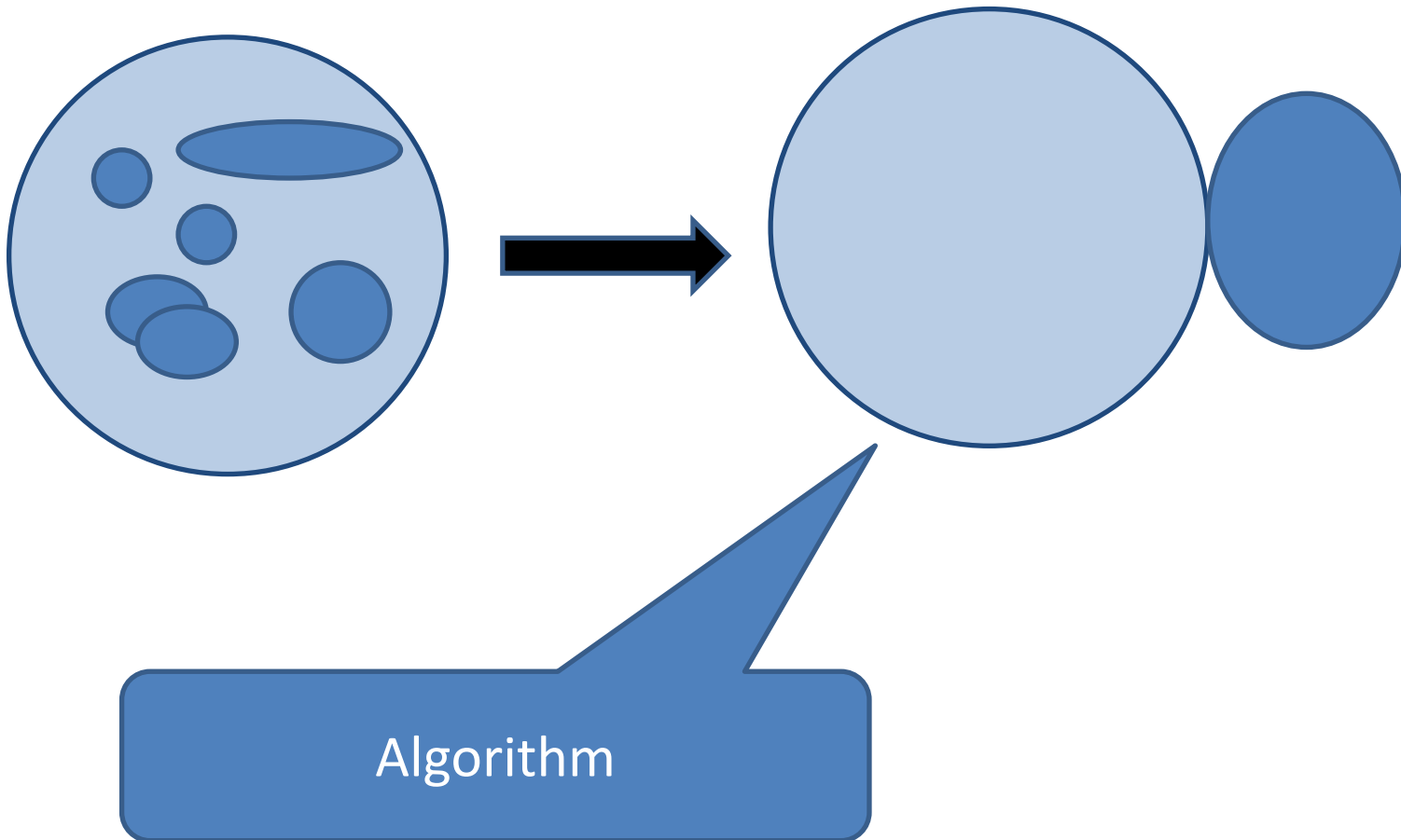
- deal with threads explicitly

# BUT

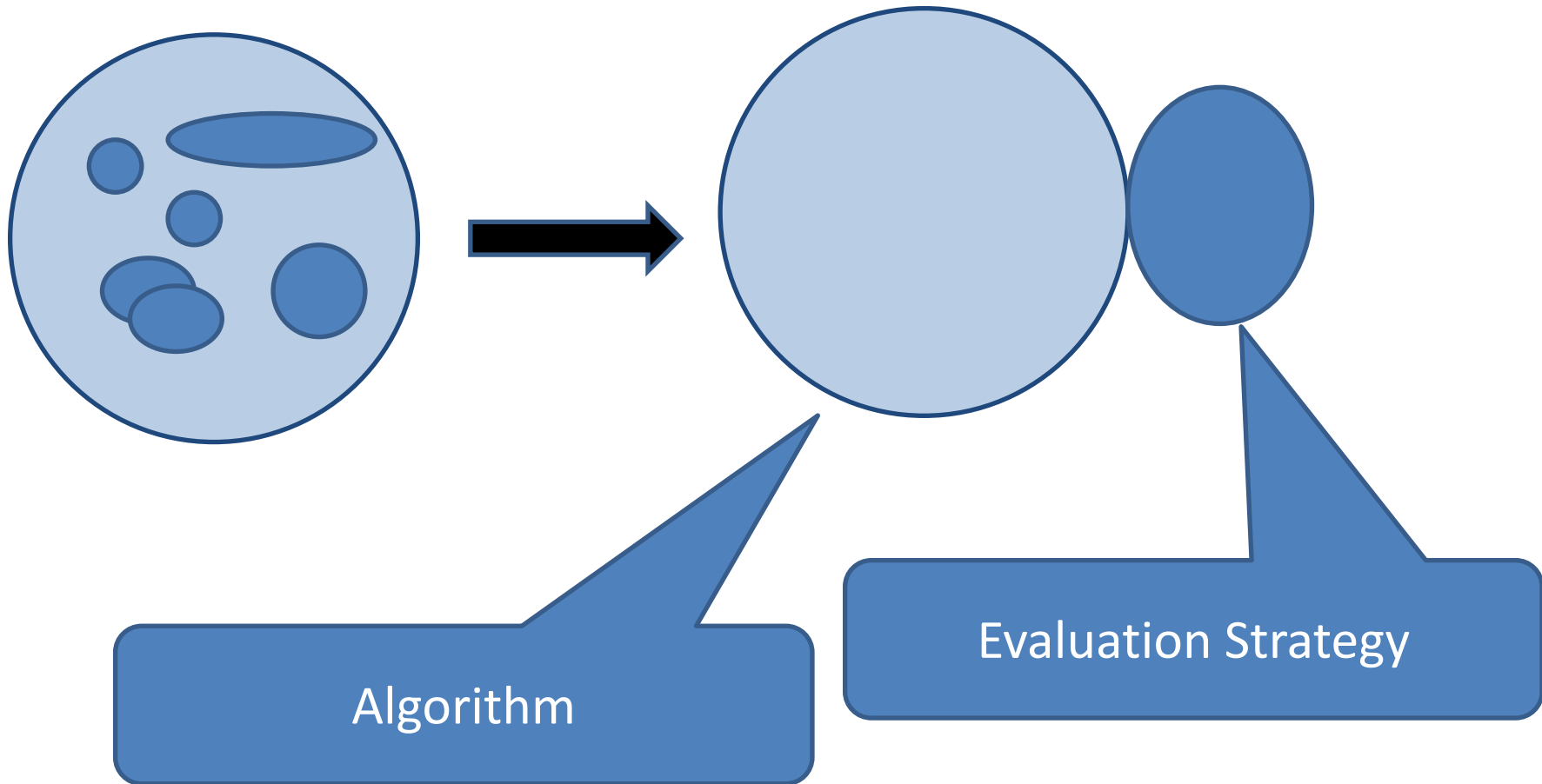


Original code + par + pseq + rnf etc.  
can be opaque

# Separate concerns



# Separate concerns



# Evaluation Strategies

express dynamic behaviour independent of the algorithm

provide abstractions above par and pseq

are modular and compositional  
(they are ordinary higher order functions)

can capture patterns of parallelism



# Papers

## *Algorithm + Strategy = Parallelism*

P.W. TRINDER

*Department of Computing Science, University of Glasgow, Glasgow, UK*

K. HAMMOND

*Division of Computing Science, University of St Andrews, St Andrews, UK*

H.-W. LOIDL AND S.L. PEYTON JONES <sup>†</sup>

*Department of Computing Science, University of Glasgow, Glasgow, UK*

JFP 1993

## **Seq no more: Better Strategies for Parallel Haskell**

Simon Marlow

Microsoft Research, Cambridge, UK  
simonmar@microsoft.com

Patrick Maier

Heriot-Watt University, Edinburgh, UK  
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh, UK  
H.W.Loidl@hw.ac.uk

Haskell'10

Mustafa K. Aswad

Heriot-Watt University, Edinburgh, UK  
mka10@hw.ac.uk

Phil Trinder

Heriot-Watt University, Edinburgh, UK  
P.W.Trinder@hw.ac.uk

# Papers

*Algorithm + Strategy = Parallelism*

P.W. TRINDER

*Department of Computing Science, University of Glasgow*

K. HAMMOND

*Division of Computing Science, University of St Andrews*

H.-W. LOIDL AND S.L. PEYTON JONES

*Department of Computing Science, University of Glasgow*

Redesigns strategies

richer set of parallelism combinators

Better specs (evaluation order)

Allows new forms of coordination

generic regular strategies over data structures

speculative parallelism

monads everywhere 😊

Presentation is about New Strategies

**Seq no more: Better Parallelism**

Simon Marlow

Microsoft Research, Cambridge, UK  
simonmar@microsoft.com

Patrick Maier

Heriot-Watt University, Edinburgh, UK  
P.Maier@hw.ac.uk

Heriot-Watt University, Edinburgh, UK  
H.W.Loidl@hw.ac.uk

FRASER 10

Mustafa K. Chewad

Heriot-Watt University, Edinburgh, UK  
mka10@hw.ac.uk

Phil Trinder

Heriot-Watt University, Edinburgh, UK  
P.W.Trinder@hw.ac.uk

# The Eval monad

```
import Control.Parallel.Strategies

data Eval a
instance Monad Eval

runEval :: Eval a -> a

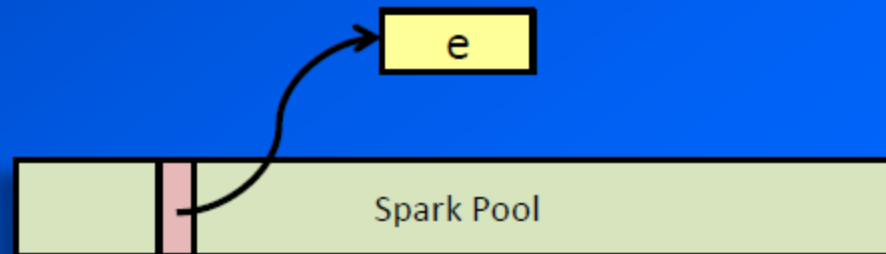
rpar :: a -> Eval a
rseq :: a -> Eval a
```

- Eval is pure
- Just for expressing sequencing between rpar/rseq – nothing more
- Compositional – larger Eval sequences can be built by composing smaller ones using monad combinators
- Internal workings of Eval are very simple (see Haskell Symposium 2010 paper)

# What does `rpar` actually do?

```
x <- rpar e
```

- `rpar` creates a *spark* by writing an entry in the *spark pool*
  - `rpar` is very cheap! (not a thread)
- the spark pool is a circular buffer
- when a processor has nothing to do, it tries to remove an entry from its own spark pool, or steal an entry from another spark pool (*work stealing*)
- when a spark is found, it is evaluated
- The spark pool can be full – watch out for spark overflow!



# Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```

# Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-1))
    return (nf1 + nf2 + 1)
```



do this  
spark nfib (n-1)

# Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```

and **then** this  
eval `nfib(n-2)` and  
wait for result

# Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```

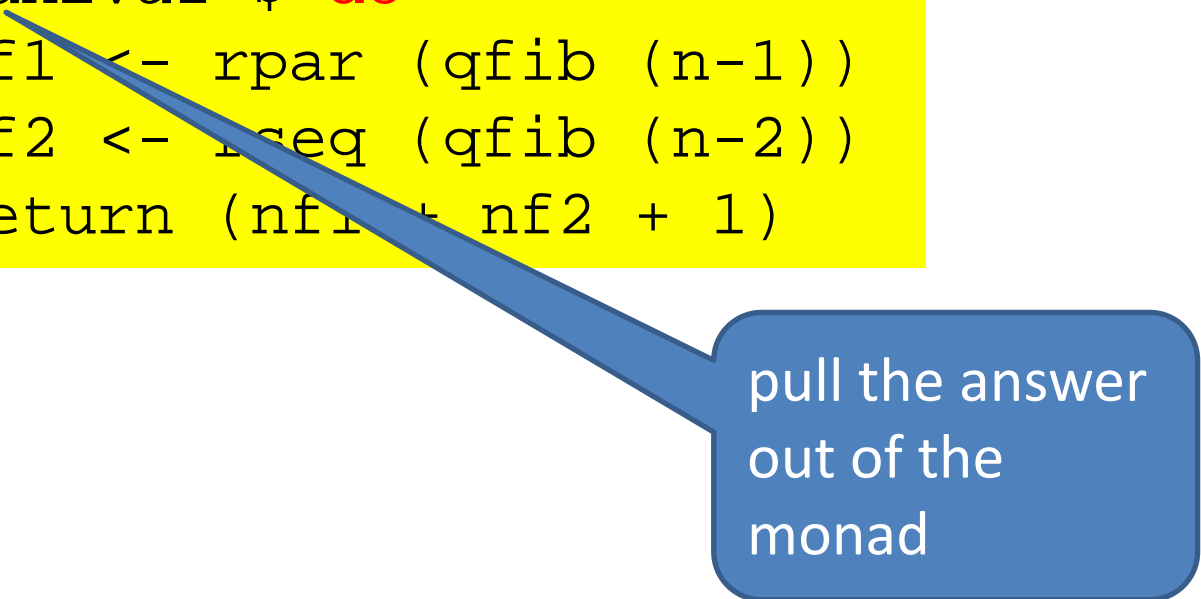


the result



# Expressing evaluation order

```
qfib :: Integer -> Integer
qfib n | n < 2 = 1
qfib n = runEval $ do
    nf1 <- rpar (qfib (n-1))
    nf2 <- rseq (qfib (n-2))
    return (nf1 + nf2 + 1)
```



pull the answer  
out of the  
monad

# What do we have?

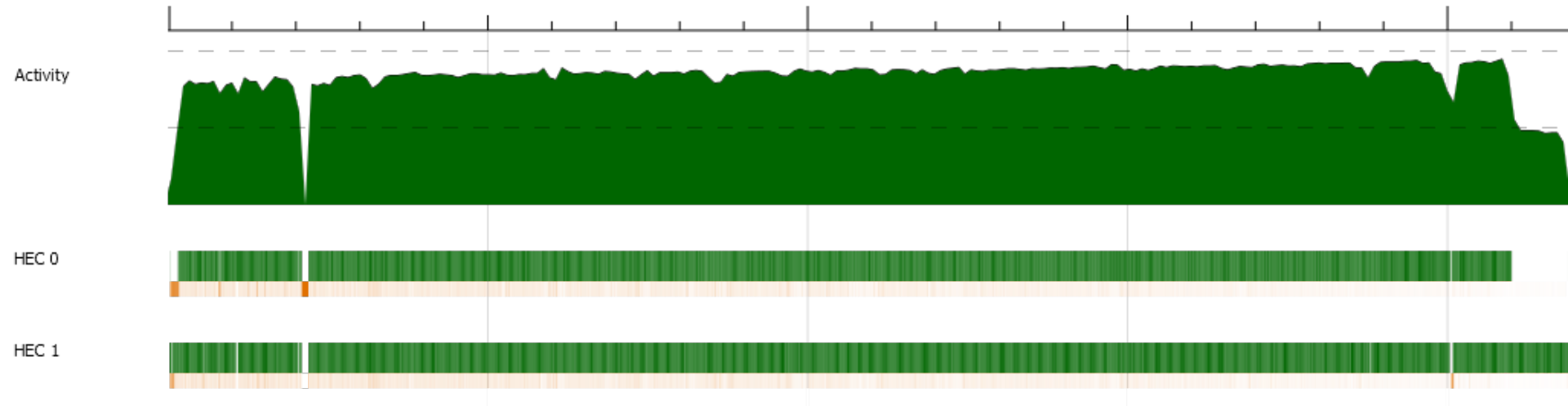
The Eval monad raises the level of abstraction for pseq and par; it makes fragments of evaluation order first class, and lets us compose them together. We should think of the Eval monad as an Embedded Domain-Specific Language (EDSL) for expressing evaluation order, embedding a little evaluation-order constrained language inside Haskell, which does not have a strongly-defined evaluation order.

(from Haskell 10 paper)

# parallel map

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do
    b  <- rpar (f a)
    bs <- parMap f as
    return (b:bs)
```

# Using parMap

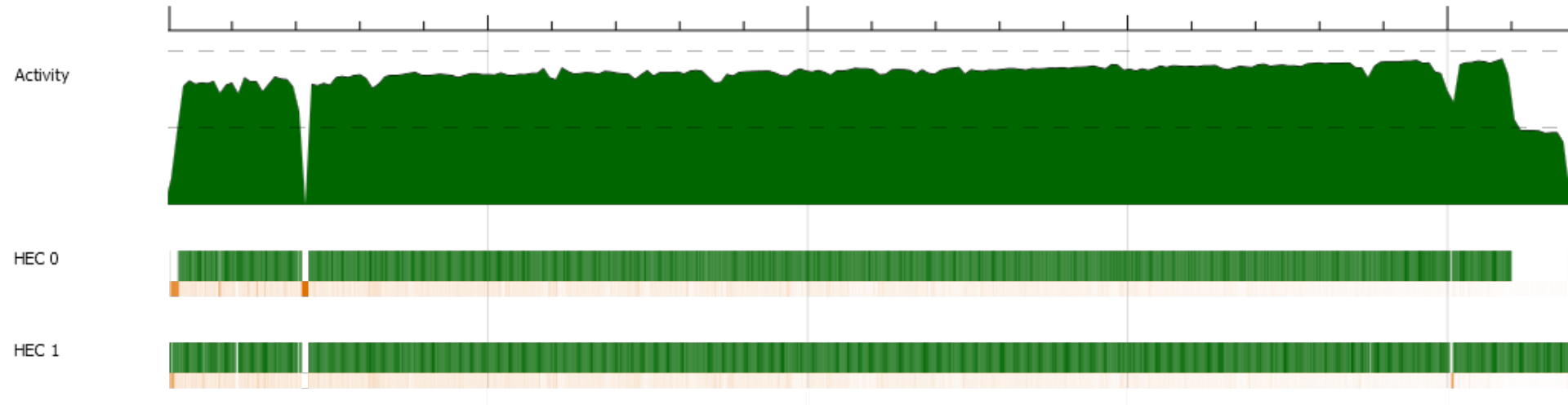


```
foo :: Integer -> Integer  
foo = \a -> sum [1 .. a]
```

```
print $ sum $ runEval $ (parMap foo (reverse [1..10000]))
```

SPARKS: 10000 (8191 converted, 0 pruned)

# Using our parMap



```
foo :: Integer -> Integer  
foo = \a -> sum [1 .. a]
```

```
print $ sum $ runEval $ (parMap foo reverse [1..10000]))
```

```
SPARKS 10000 (8191 c
```

#sparks =  
length of list

# parallel map

- + Captures a pattern of parallelism
- + good to do this for standard higher order function like map
- + can easily do this for other standard sequential patterns

return (b.bs)

# BUT

- had to write a new version of map
- mixes algorithm and dynamic behaviour



return (b.bs)

# Evaluation Strategies



Raise level of abstraction

Encapsulate parallel programming idioms as reusable components that can be composed



# Strategy (as of 2010)

```
type Strategy a = a -> Eval a
```

function

evaluates its input to some degree

traverses its argument and uses `rpar` and `rseq` to express dynamic behaviour / sparking

returns an equivalent value in the `Eval` monad

# using

```
using :: a -> Strategy a -> a  
x `using` s = runEval (s x)
```

Program typically applies the strategy to a structure and then uses the returned value, discarding the original one (which is why the value had better be equivalent)

An almost identity function that does some evaluation and expresses how that can be parallelised

# Basic strategies

```
r0 :: Strategy a
```

```
r0 x = return x
```

```
rpar :: Strategy a
```

```
rpar x = x `par` return x
```

```
rseq :: Strategy a
```

```
rseq x = x `pseq` return x
```

```
rdeepseq :: NFData a => Strategy a
```

```
rdeepseq x = rnf x `pseq` return x
```

# Basic strategies

```
r0 :: Strategy a  
r0 x = return x
```

NO evaluation

```
rpar :: Strategy a  
rpar x = x `par` return x
```

```
rseq :: Strategy a  
rseq x = x `pseq` return x
```

```
rdeepseq :: NFData a => Strategy a  
rdeepseq x = rnf x `pseq` return x
```

# Basic strategies

```
r0 :: Strategy a  
r0 x = return x
```

```
rpar :: Strategy a  
rpar x = x `par` return x
```

```
rseq :: Strategy a  
rseq x = x `pseq` return x
```

```
rdeepseq :: NFData a => Strategy a  
rdeepseq x = rnf x `pseq` return x
```



spark x

# Basic strategies

```
r0 :: Strategy a  
r0 x = return x
```

```
rpar :: Strategy a  
rpar x = x `par` return x
```

```
rseq :: Strategy a  
rseq x = x `pseq` return x
```

```
rdeepseq :: NFData a => Strategy a  
rdeepseq x = rnf x `pseq` return x
```



evaluate x  
to WHNF

# Basic strategies

```
r0 :: Strategy a  
r0 x = return x
```

```
rpar :: Strategy a  
rpar x = x `par` return x
```

```
rseq :: Strategy a  
rseq x = x `pseq` return x
```

```
rdeepseq :: NFData a => Strategy a  
rdeepseq x = rnf x `pseq` return x
```



fully evaluate x

# parList

```
parList :: Strategy a -> Strategy [a]
```

```
parList strat (x:xs) = do  
  x'   <- rpar (x `using` strat )  
  xs'  <- parList strat xs  
  return (x' : xs')
```



# parList

```
parList :: Strategy a -> Strategy [a]
```

```
parList strat (x : xs) = do
```

```
  x' <- rpar (x) (parList strat xs)
```

```
  xs' <- rpar (xs) (parList strat xs)
```

```
  return (x' : xs')
```

Takes a Strategy on a and returns a Strategy on lists of a  
Building strategies from smaller ones

# parList

```
parList :: Strategy a -> Strategy [a]

parList strat (x:xs) = do
  x'  <- rpar (x `using` strat )
  xs' <- parList strat xs
  return (x' : xs')
```

Return a value because otherwise the runtime would happily get rid of a spark that seems unwanted (see paper)

Remember not to drop return value of rpar

# evalList

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

# evalList

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
                       xs' <- evalList s xs
                       return (x':xs')
```

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

# evalList (more general)

```
evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x : xs) = do x' <- s x
```

The equivalent of evalList is available for many data structures (Traversable). So we defining parX for many X is really easy

```
parList
parList
```

=> generic strategies for data-oriented parallelism



# another list strategy

```
parListSplitAt :: Int -> Strategy [a] -> Strategy [a]  
               -> Strategy [a]
```

```
parListSplitAt n stratL stratR
```

n

par

stratL

stratR

# How do we *use* a Strategy?

---

```
type Strategy a = a -> Eval a
```

- We could just use runEval
- But this is better:

```
x `using` s = runEval (s x)
```

- e.g.

```
myList `using` parList rdeepseq
```

- Why better? Because we have a “law”:
  - $x \text{ `using` } s \approx x$
  - We can insert or delete “using s” without changing the semantics of the program

# Is that really true?

---

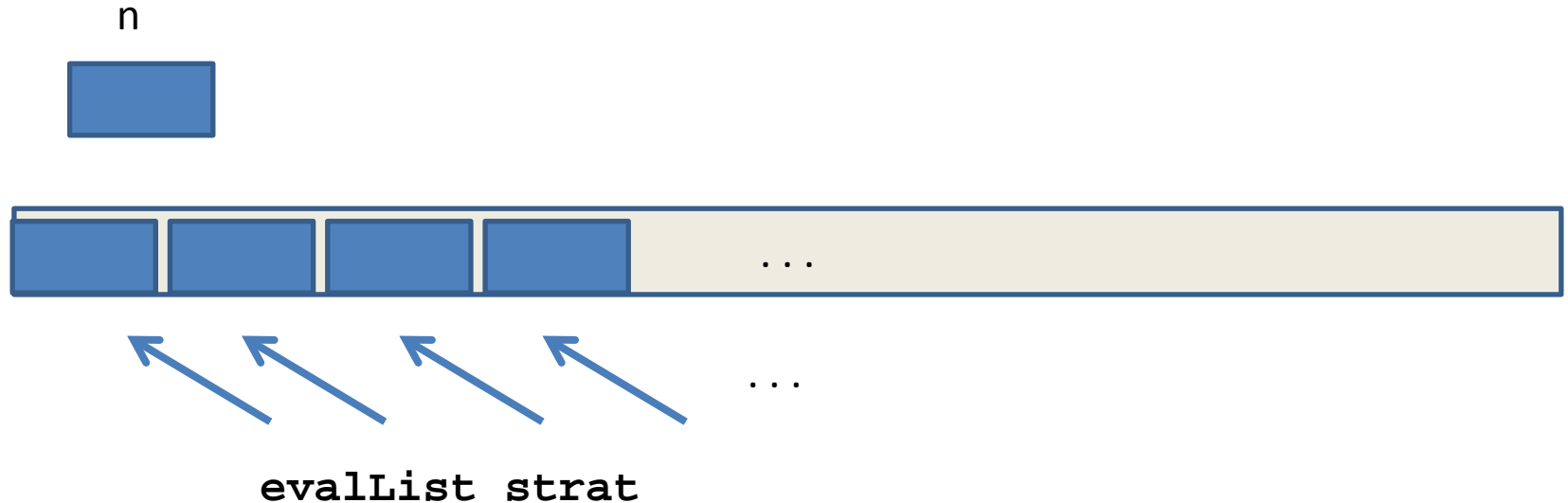
- Well, not entirely.
  1. It relies on Strategies returning “the same value” (*identity-safety*)
    - Strategies from the library obey this property
    - Be careful when writing your own Strategies
  2. `x`using`s` might do more evaluation than just `x`.
    - So the program with `x`using`s` might be `_|_`, but the program with just `x` might have a value
- if identity-safety holds, adding `using` cannot make the program produce a different result (other than `_|_`)



# using yet another list strategy

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

```
parListChunk n strat
```



# using yet another list strategy

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $ runEval $ parMap foo (reverse [1..10000])
```

Now

```
print $ sum $  
(map foo (reverse [1..10000])) `using` parListChunk 50 rdeepseq )
```

SPARKS: 200 (200 converted, 0 pruned)

```
parListChunk :: Int -> Strategy a -> Strategy [a]
```

Before

```
print $ sum $
```

Now

```
print $ sum $  
(map foo (reverse [0..10000])) `using` parListChunk 50 rdeepseq )
```

Remember not to be a control freak, though. Generating plenty of sparks gives the runtime the freedom it needs to make good choices (=> Dynamic partitioning for free)

SPARKS: 200 (200 converted, 0 pruned)

# using is not always what we need

- Trying to pull apart algorithm and coordination in `qfib` (from earlier) doesn't really give a satisfactory answer (see Haskell 10 paper)

(If the worst comes to the worst, one can get explicit control of threads etc. in concurrent Haskell, but determinism is lost... )

# Divide and conquer

Capturing patterns of parallel computation is a major strong point of strategies

D&C is a typical example (see also parBuffer, parallel pipelines etc.)

```
divConq :: (a -> b)
         -> a
         -> (a -> Bool)
         -> (b -> b -> b)
         -> (a -> Maybe (a,a))
         -> b
```

function on base cases  
input  
par threshold reached?  
combine  
divide  
result

# Divide and Conquer

```
divConq f arg threshold combine divide = go arg
  where
    go arg =
      case divide arg of
        Nothing      -> f arg
        Just (l0,r0) -> combine l1 r1 `using` strat
          where
            l1 = go l0
            r1 = go r0
            strat x = do r l1; r r1; return x
                  where r | threshold arg = rseq
                          | otherwise     = rpar
```

Separates algorithm and strategy

A first inkling that one can probably do interesting things by programming with strategies

# Using D&C

```
pscan t op as = divConq f as threshold combine divide
where
```

```
  f = scanl1 op
```

```
  threshold ls = length ls < t
```

```
  combine ls rs = ls ++ [op (last ls) b | b <- rs]
```

```
  divide []      = Nothing
```

```
  divide [a]     = Nothing
```

```
  divide [a,b]  = Nothing
```

```
  divide ls     = Just $ splitAt (div (length ls + 1) 2) ls
```

Q: What does this do??

Note that it is EMBARASSINGLY SLOW. How can this be improved?

# Skeletons

- encode fixed set of common coordination patterns and provide efficient parallel implementations (Cole, 1989)
- Popular in both functional and non-functional languages. See particularly Eden (Loogen et al, 2005)

A difference: one can / should roll ones own strategies

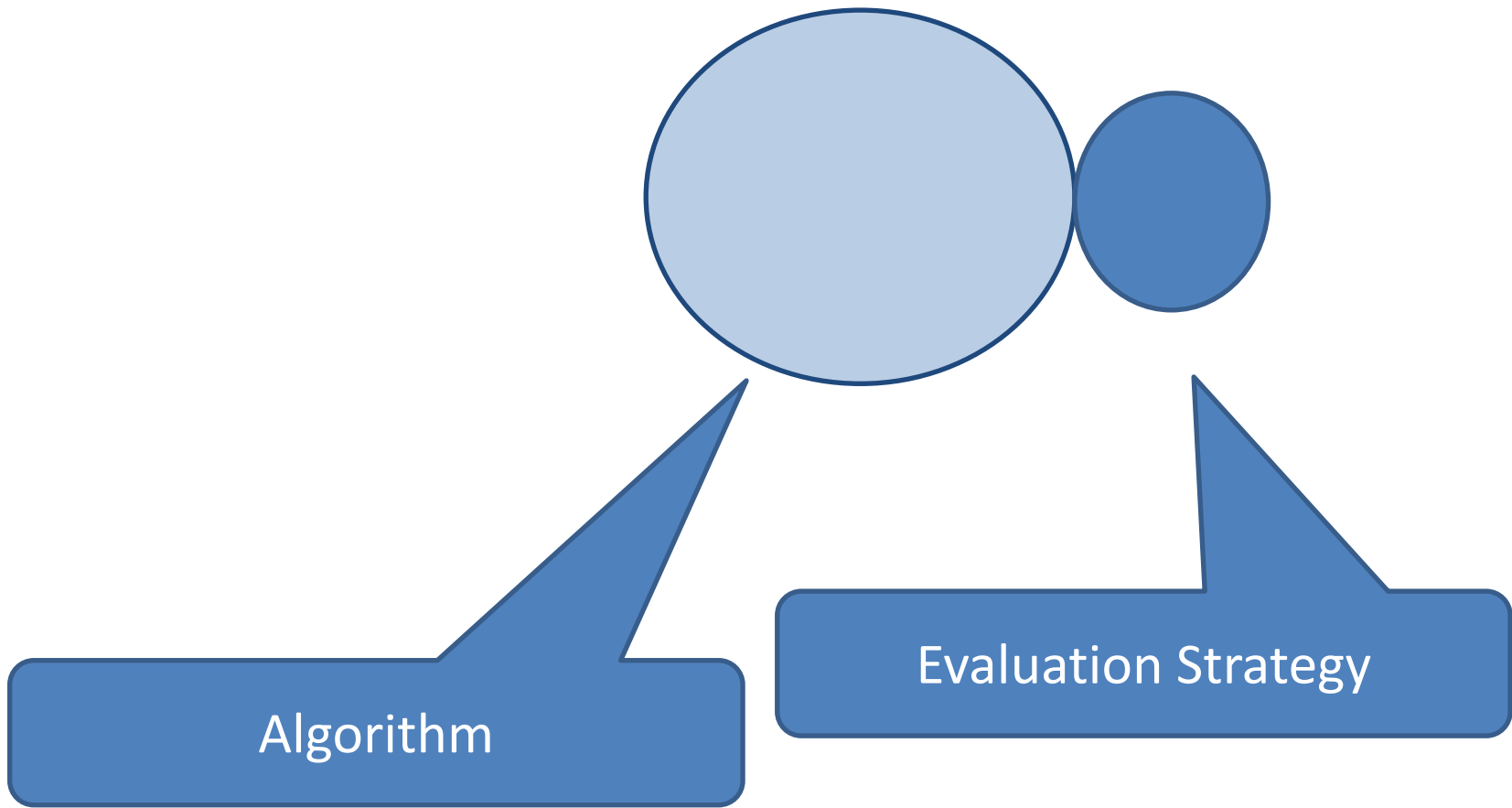
- Will go into skeletons and their relation to strategies in Jost Berthold's lecture (L2 of W3)



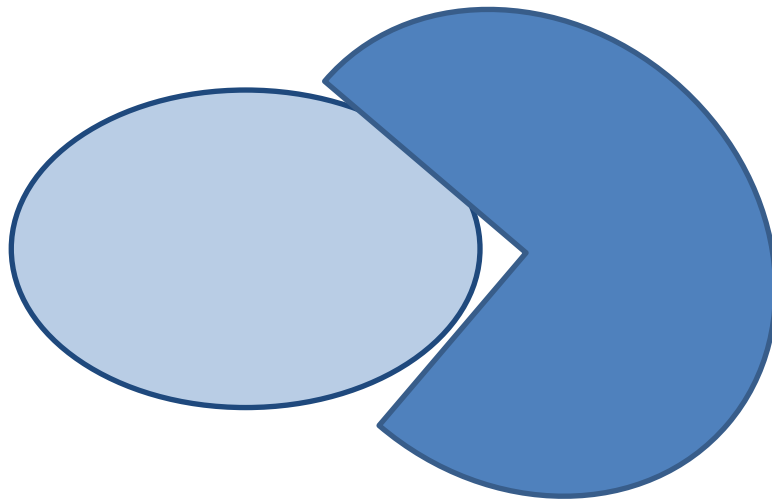
# Strategies: summary

- + elegant redesign by Marlow et al (Haskell 10)
- + better separation of concerns
- + Laziness is essential for modularity
- + generic strategies for (Traversable) data structures
- + Marlow's CEFPP notes contain a nice **kmeans** example, but note the need to redefine the algorithm to make it parallelisable....
- Having to think so much about evaluation order is worrying!  
Laziness is not only good here.

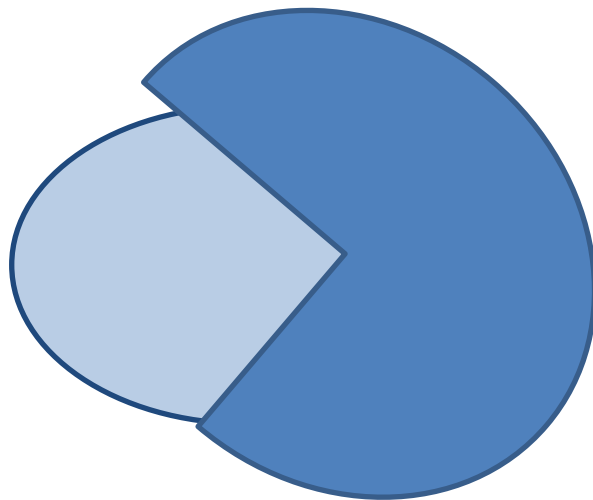
# Strategies: summary



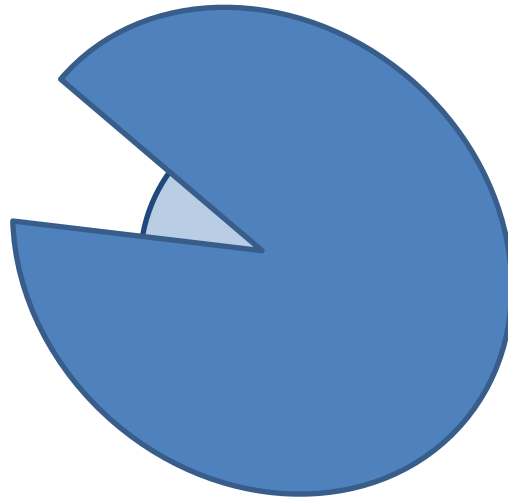
# Better visualisation

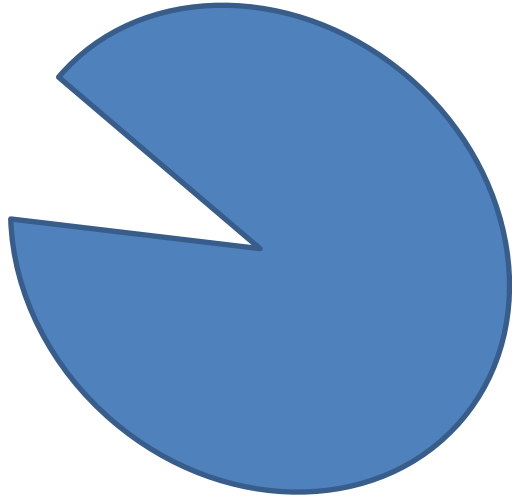
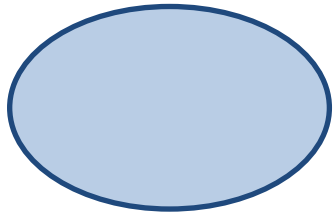


# Better visualisation



# Better visualisation





# When strategies don't cut it

Often in more control oriented parallelism

Simon Marlow:      the Par Monad

Next **Wednesday's** lecture will be by the man himself (in E-Studio, with exercises)

# Next lecture

Monday : Andres Löh from Well-Typed on  
Threadscope and GHC Events

Don't miss it! He wants you to do exercises, so  
try to get some practice beforehand.



# Just for fun



- For the fastest Haskell sorter
- Using `par`, `pseq`, but *no mutable data*
- Sorting the random list used in this lecture
- On a 4-core Intel i7 with 4 HECs
- **Entries:** on Monday 19 March via Fire system