Going bananas

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

Functional Programming with Bananas, ~~Lenses,~~ ~~Envelopes and Barbed Wire~~

Erik Meijer * Maarten Fokkinga † Ross Paterson ‡

Functional Programming Languages and Computer Architecture 1991 (FPCA'91)

Reading this paper causes serious headache.

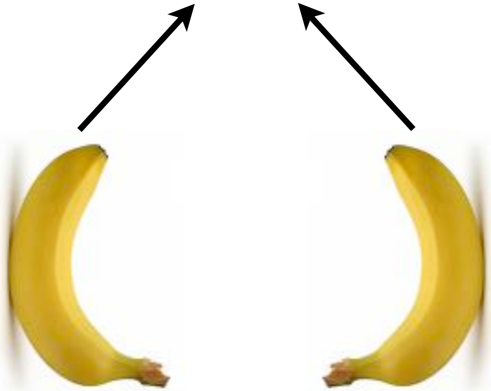
Catamorphisms

Let $b \in B$ and $\oplus \in A \| B \to B$, then a list-catamorphism $h \in A* \to B$ is a function of the following form:

$$\begin{aligned} h\ \mathrm{Nil} &= b \\ h\ (\mathrm{Cons}\ (a, as)) &= a \oplus (h\ as) \end{aligned} \tag{1}$$

In the notation of Bird&Wadler [5] one would write $h = \mathrm{foldr}\ b\ (\oplus)$. We write catamorphisms by wrapping the relevant constituents between so called banana brackets:

$$h = (\!| b, \oplus |\!) \tag{2}$$



foldr trivia

```
> let l1 = [1,2,3,4]
> length l1
4

> sum l1
10

length :: [a] -> Int
length = foldr (const (+1)) 0

sum :: Num a => [a] -> a
sum = foldr (+) 0

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
```

mapping with **foldr**

```
> let l1 = [1,2,3,4]
> map (+1) l1
[2,3,4,5]

> reverse l1
[4,3,2,1]

map :: (a -> b) -> [a] -> [b]
map f = foldr ((:) . f) []

reverse :: [a] -> [a]
reverse = foldr (\x xs -> xs ++ [x]) []
```

More **foldr**-based traversal

```
> let l1 = [1,2,3,4]
> filter odd l1
[1,3]

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr f []
 where f x = if p x then (:) x else id

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v xs = foldr h id xs v
 where h = (\x g -> (\a -> g (f a x)))
```

foldrs with effects

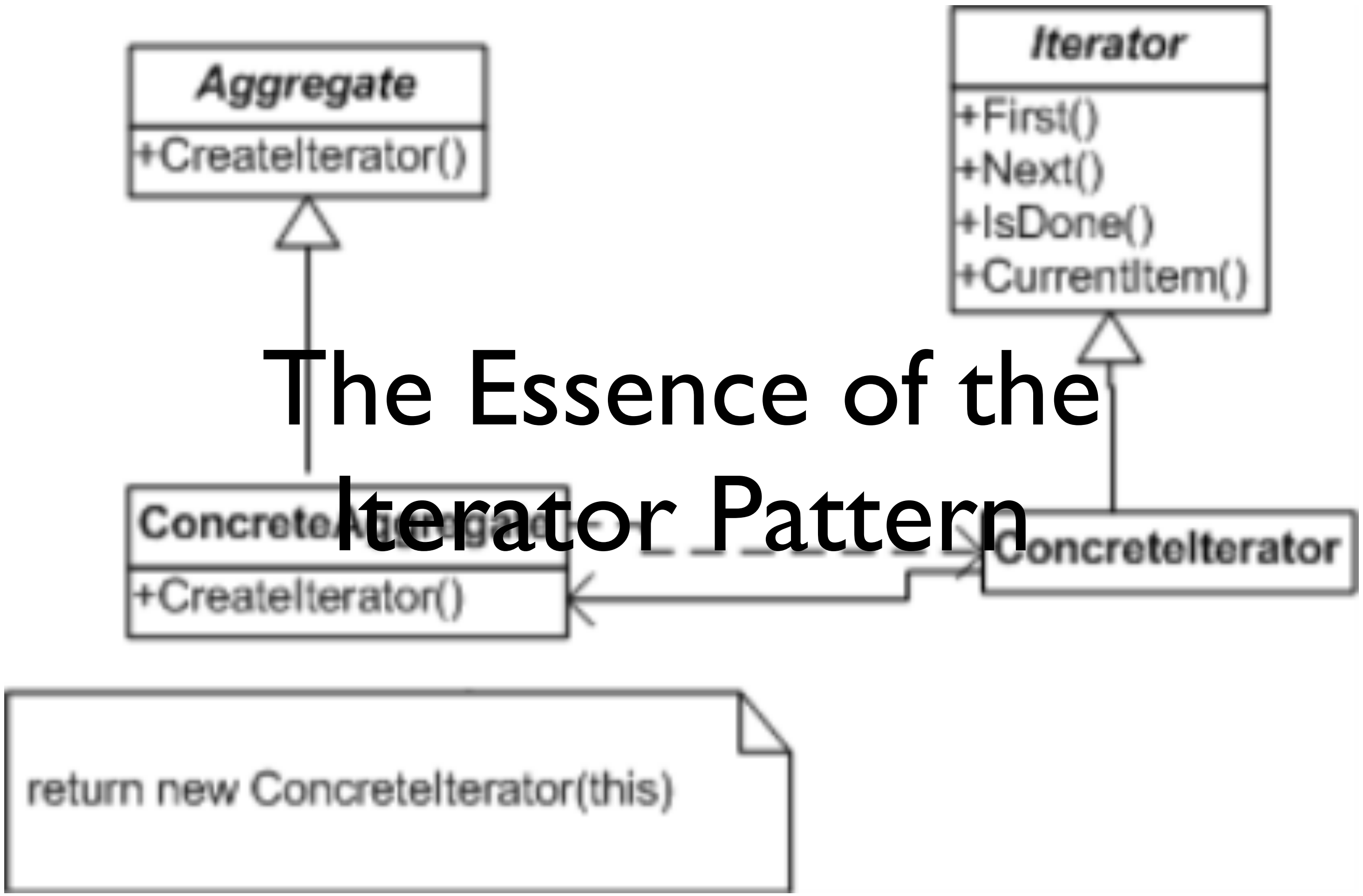
```
> let l1 = [1,2,3,4]
> runState (mapM (tick (+1)) l1) 0
([2,3,4,5],4)

tick :: (x -> y) -> x -> State Int y
tick f x = get >>= put . (+1) >> return (f x)

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM g = foldr f k
 where
  f x mys = do y <- g x; ys <- mys; return (y:ys)
  k = return []
```

Works too with “modern” effects: *applicative functors*.

Some million $ questions

How to ...

- ... traverse *abstract* lists?
- ... traverse data in *parallel*?
- ... traverse *heterogenous* data?
- ... scrap your boilerplate?



The Essence of the Iterator Pattern

Functors: datatypes that can be *mapped* in a *shape-preserving* manner.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [ ] where
  fmap = map

-- laws
fmap id = id
fmap (p . q) = (fmap p) . (fmap q)
```

Foldables: datatypes that can be *folded* with shape extinction & *aggregation*.

```
class Foldable t where
 fold      :: Monoid m => t m -> m
 foldMap   :: Monoid m => (a -> m) -> t a -> m
 foldr     :: (a -> b -> b) -> b -> t a -> b
 foldl     :: (a -> b -> a) -> a -> t b -> a
 foldr1    :: (a -> a -> a) -> t a -> a
 foldl1    :: (a -> a -> a) -> t a -> a
```

Either of foldMap or foldr is sufficient for a minimally complete definition. ***Exercise***: suggest suitable defaults for the others.

Monoids: datatypes for results of folding.

```
class Monoid a where
  mempty  :: a               -- identity
  mappend :: a -> a -> a     -- associative op.

newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x `mappend` Sum y = Sum (x + y)

instance Num a => Monoid (Product a) where
  mempty = Product 1
  Product x `mappend` Product y = Product (x * y)
```

Traversables: datatypes that can be traversed from left to right.

```
class (Functor t, Foldable t) => Traversable t where
  traverse    :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA   :: Applicative f => t (f a) -> f (t a)
  mapM        :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence    :: Monad m => t (m a) -> m (t a)
```

traverse is really ***the*** key operation.
Think of it as a monadic map--except that we use applicative functors instead of monads.

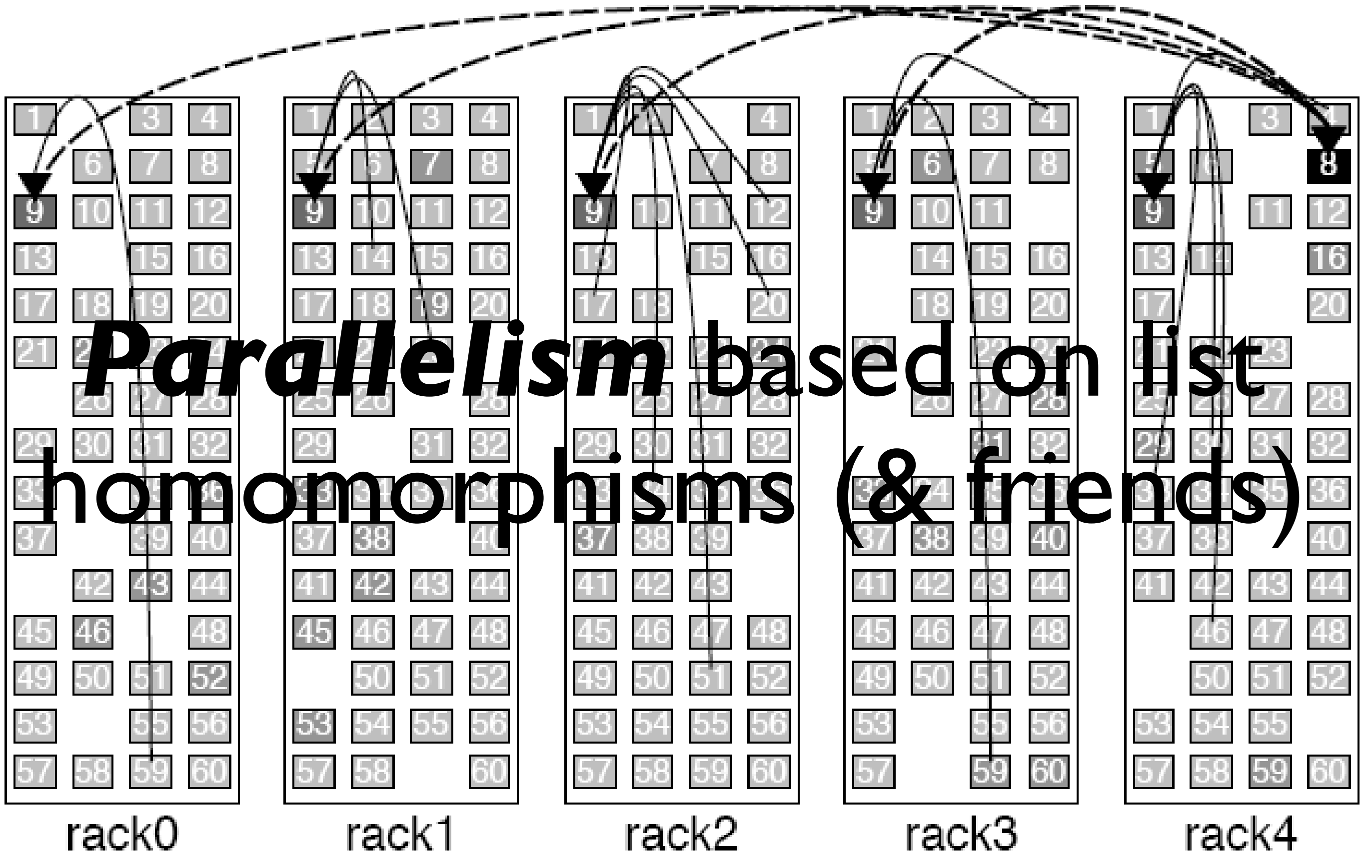
Applicative Functors: a more applicative form of effects, when compared to monads.

```
> let l1 = [1,2,3,4]
> runState (traverse (tick ((+) (-1))) l1) 0
([0,1,2,3],4)

tick :: (x -> y) -> x -> State Int y
tick f x = get >>= put . (+1) >> return (f x)

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative (State s) where
  pure = return
  mf <*> mx = mf >>= \f -> mx >>= return . f
```

Properties omitted here

http://en.wikibooks.org/wiki/Haskell/Applicative_Functors

Parallelism based on list homomorphisms (& friends)



http://labs.google.com/papers/sawzall.html

A function h :: [a] -> b is a list homomorphisms if there is a function f :: a -> b, an associative operation o :: b -> b with unit u :: b such that the following equations hold:

$$\begin{aligned} h\ [\ ] &= u \\ h\ [x] &= f\ x \\ h\ (xs \mathbin{+\!+} ys) &= h\ xs\ \text{`}o\text{`}\ h\ ys \end{aligned}$$

The *wordOccurrenceCount* example

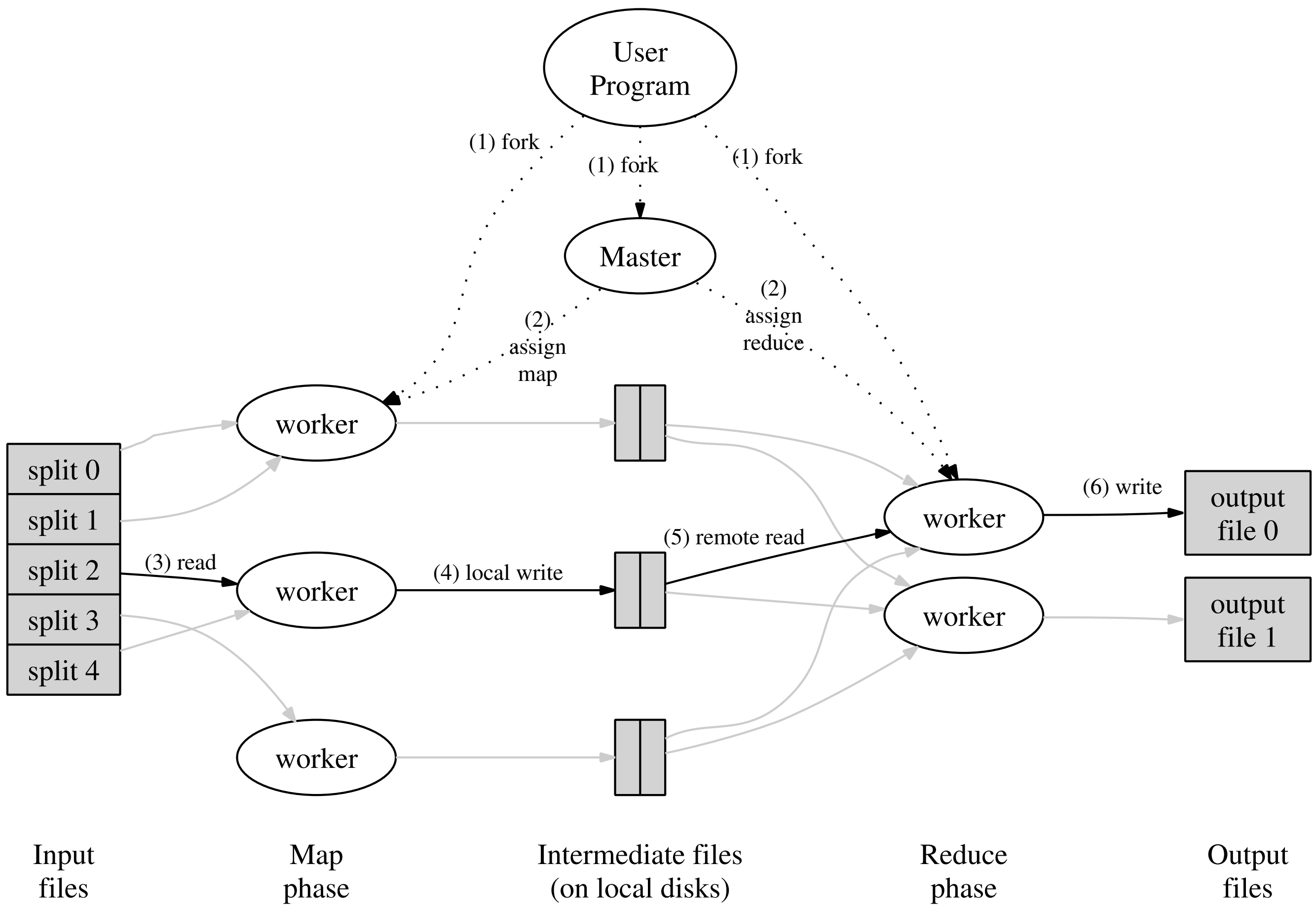
```
main =
        print
    $   wordOccurrenceCount
    $   insert "doc2" "appreciate the unfold"
    $   insert "doc1" "fold the fold"
    $   empty
```

Petabytes of website content

```
> main
[("appreciate",1),("fold",2),("the",2),("unfold",1)]
```

List each word with its occurrence count

Google's MapReduce



[http://labs.google.com/papers/mapreduce.html]

Google code for *wordOccurrenceCount*

```
map(String key, String value):
 // key: document name
 // value: document contents
 for each word w in value:
  EmitIntermediate(w, "1");
```

Ignore doc key

Use word as intermediate key

```
reduce(String key, Iterator values):
 // key: a word
 // values: a list of counts
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

Implicitly preserve word key

[http://labs.google.com/papers/mapreduce.html]

Haskell code for *wordOccurrenceCount*

```
wordOccurrenceCount = mapReduce m r
 where

 m :: String -> String -> [(String,Int)]
 m = const (map (flip (,) 1) . words)

 r :: String -> [Int] -> Int
 r = const sum
```

Reusable skeleton of MapReduce computations

Split up doc string into words

Pair up each word with 1

Ignore doc key

Sum up distributed counts of words

Ignore word

A specification of MapReduce

```
mapReduce  ::  forall k1 k2 v1 v2. Ord k2
           =>  (k1 -> v1 -> [(k2,v2)])        -- "map"
           ->  (k2 -> [v2] -> v2)             -- "reduce"
           ->  Map k1 v1 ->  Map k2 v2        -- I/O

mapReduce m r = reducePerKey . groupByKey . mapPerKey
 where

  mapPerKey :: Map k1 v1 -> [(k2,v2)]
  mapPerKey = concat . map (uncurry m) . toList

  groupByKey :: [(k2,v2)] -> Map k2 [v2]
  groupByKey = foldl insert empty
   where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict

  reducePerKey :: Map k2 [v2] -> Map k2 v2
  reducePerKey = mapWithKey r
```

Exercise: find arguments *m* and *r* such that the number of words per document is counted in addition to just the word occurrences.

A ***parallel*** model of MapReduce

```
mapReduce'  ::  forall k1 k2 v1 v2. Ord k2
         =>  Int                              -- Number of reducers
         ->  (k2 -> Int)                      -- Reducer association
         ->  (k1 -> v1 -> [(k2,v2)])          -- "map"
         ->  (k2 -> [v2] -> v2)               -- "reduce"
         ->  [Map k1 v1] ->  [Map k2 v2]      -- I/O

mapReduce' n a m r
 =    map ( reducePerKey . mergeByKey )
 .    transpose
 .    map (      map ( reducePerKey . groupByKey )
            .    partion
            .    mapPerKey )
 where
  partion :: [(k2,v2)] -> [[(k2,v2)]]
  partion y = map (\k -> filter ((==) k . a . fst) y) [1.. n]

  mergeByKey :: [Map k2 v2] -> Map k2 [v2]
  mergeByKey = unionsWith (++) . map (mapWithKey (\_ v2 -> [v2]))

  mapPerKey :: Map k1 v1 -> [(k2,v2)]
  groupByKey :: [(k2,v2)] -> Map k2 [v2]
  reducePerKey :: Map k2 [v2] -> Map k2 v2
```

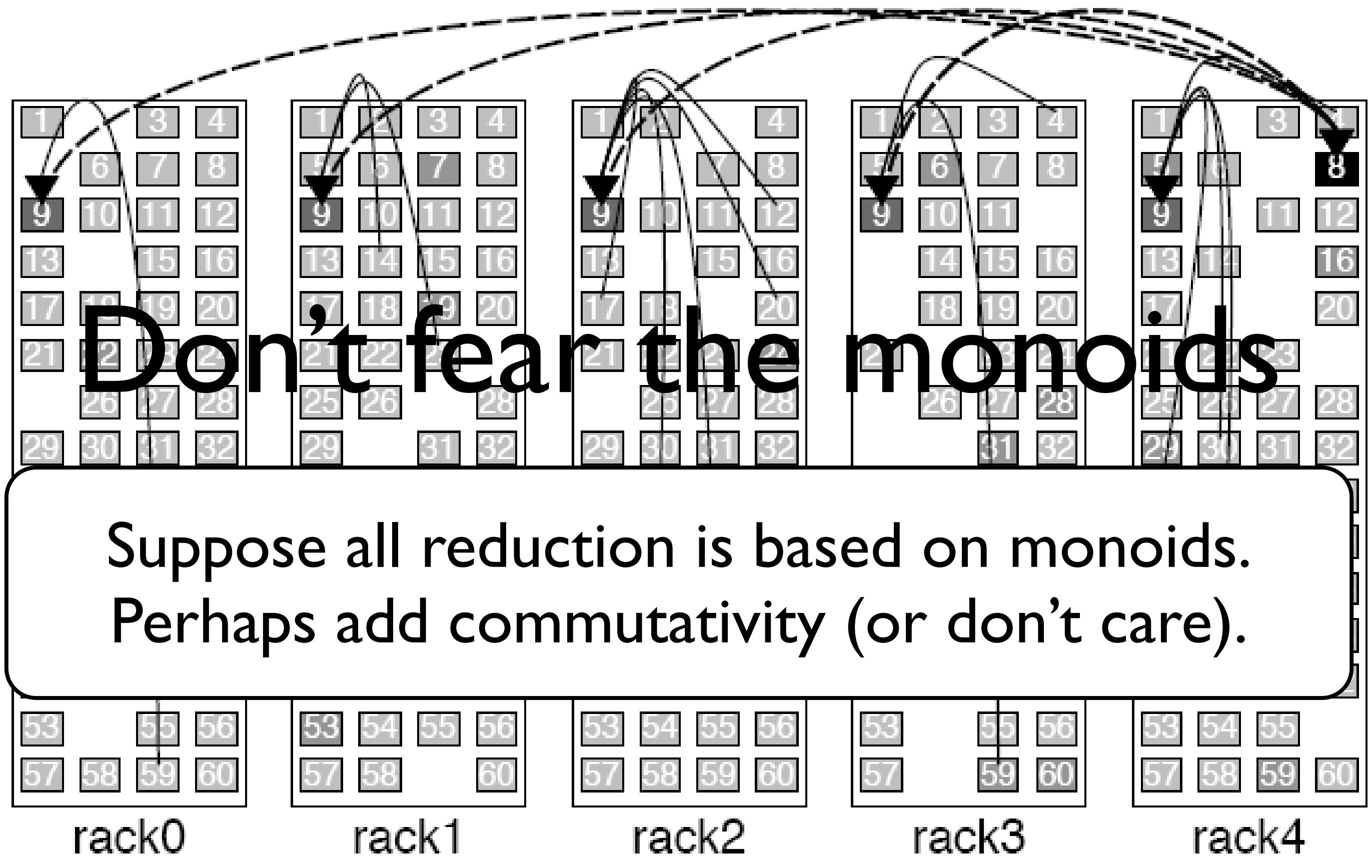
parallel map

communication

parallel reduce

as before

Don’t fear the monoids



http://labs.google.com/papers/sawzall.html

Suppose all reduction is based on monoids.
Perhaps add commutativity (or don’t care).

```
mapReduce      :: Monoid m => (x -> m) -> [x] -> m
mapReduce f    = foldr (mappend . f) mempty

mapReduce'     :: Monoid m => (x -> m) -> [[x]] -> m
mapReduce' f   = mconcat . map (mapReduce f)

mapReduce''    :: Monoid m => (x -> m) -> [[[x]]] -> m
mapReduce'' f  = mconcat . map (mapReduce' f)
```

For example:
what’s the monoid
for counting word
occurrences?

The monoid for *wordOccurrenceCount*

```
newtype (Ord k, Monoid v) =>
        MapToMonoid k v =
        MapToMonoid { getMap :: Map k v }
```

We need to "override" monoidal behavior of *Map*.

```
instance (Ord k, Monoid v) => Monoid (MapToMonoid k v)
 where
  mempty = MapToMonoid mempty
  mappend    (MapToMonoid f)
             (MapToMonoid g)
         =    MapToMonoid (Data.Map.unionWith mappend f g)
```

Our *Map*-level append relies on value-level append.

```
toList :: (Ord k, Monoid v) => MapToMonoid k v -> [(k,v)]
toList = Data.Map.toList . getMap

fromList :: (Ord k, Monoid v) => [(k,v)] -> MapToMonoid k v
fromList = MapToMonoid . Data.Map.fromListWith mappend
```

Monoidal code for *wordOccurrenceCount*

```
doc2words
  = MapToMonoid.fromList
  . map (flip (,) (Sum 1))
  . words
```

All programmer-provided code

```
wordOccurrenceCount =
    mapReduce doc2words
  $ map snd
  $ toList
  $  insert "doc2" "appreciate the unfold"
  $  insert "doc1" "fold the fold"
  $  empty
```

Petabytes of website content

Exercise: revisit the previous exercise and solve it in monoidal style. (That is, return maps both for word occurrences and document size.) You need monoids on pairs.

Dealing with large bananas

http://fx.worth1000.com/all-sizes/562240/banana/large

Remember those interpreters?

```
eval :: Term -> Env -> Maybe Value

eval (Var x) e = lookup x e

eval (Lambda x t) e =
  Just (InFun (\v -> eval t (insert x v e)))

eval (Apply t t') e = do
  v  <- eval t e
  v' <- eval t' e
  case v of (InFun f) -> f v'; _ -> Nothing

eval Zero _ = Just (InInt 0)
eval (Succ t) e = ... eval t e ...
eval (Pred t) e = ... eval t e ...
eval (IsZero t) e = ... eval t e ...
eval (Cond t t' t'') e = ... ...  ....
```

Exercise: try to improve the code by using applicative functors.

This interpreter is defined in compositional style. In particular, *eval* recurses into subterms.

A fold for this season

```
data TermAlgebra r
   = TermAlgebra {
  var    :: String -> r,
  lambda :: String -> r -> r,
  apply  :: r -> r -> r,
  zero   :: r,
  succ   :: r -> r,
  ...
}
```

A fold algebra for a (system of) datatype(s) is a record type with one component per constructor where references to the datatype(s) are replaced by a type parameter.

```
foldTerm :: TermAlgebra r -> Term -> r
foldTerm a = f
 where
  f (Var x) = var a x
  f (Lambda x t) = lambda a x (f t)
  f (Apply t t') = apply a (f t) (f t')
  f Zero = zero a
  f (Succ t) = succ a (f t)
  ...
```

The fold function for a (system of) datatype(s) recurses into subterms and combines the result with the algebra's component.

An interpreter based on fold

```
eval :: Term -> Env -> Maybe Value
eval = foldTerm evalAlgebra

evalAlgebra :: TermAlgebra (Env -> Maybe Value)
evalAlgebra = TermAlgebra {
  var = lookup,

  lambda = \x r e ->
    Just (InFun (\v -> r (insert x v e))),

  apply = \r r' e -> do
    v <- r e
    v' <- r' e
    case v of (InFun f) -> f v'; _ -> Nothing,

  zero = \_ -> Just (InInt 0),
  succ = ...
  pred = ...
  isZero = ...
  cond = ...
}
```

The fold algebra does not recurse by itself, neither does it refer to the underlying datatype(s)!

Another operation: analysis of free variabes

```
fv :: Term -> Set Name
fv (Var x) = singleton x
fv (Lambda x t) = delete x (fv t)
fv (Apply t t') = fv t `union` fv t'
fv Zero = empty
fv (Succ t) = fv t
fv (Pred t) = fv t
fv (IsZero t) = fv t
fv (Cond t t' t'') = fv t `union` fv t' `union` fv t''
```

Free-variable analysis based on fold

```
fv :: Term -> Set Name
fv = foldTerm fvAlgebra

fvAlgebra :: TermAlgebra (Set Name)
fvAlgebra = TermAlgebra {
  var = singleton,
  lambda = delete,
  apply = union,
  zero = empty,
  succ = id,
  pred = id,
  isZero = id,
  cond = \r r' r'' -> r `union` r' `union` r''
}
```

Observation:
essentially, sets are computed by union (except for lambda). Sets are monoids!

A monoidal fold algebra

```
malgebra :: Monoid m => TermAlgebra m
malgebra = TermAlgebra {
  var = \_ -> mempty,
  lambda = \_ r -> r,
  apply = mappend,
  zero = mempty,
  succ = id,
  pred = id,
  isZero = id,
  cond = \r r' r'' -> r `mappend` r' `mappend` r''
}
```

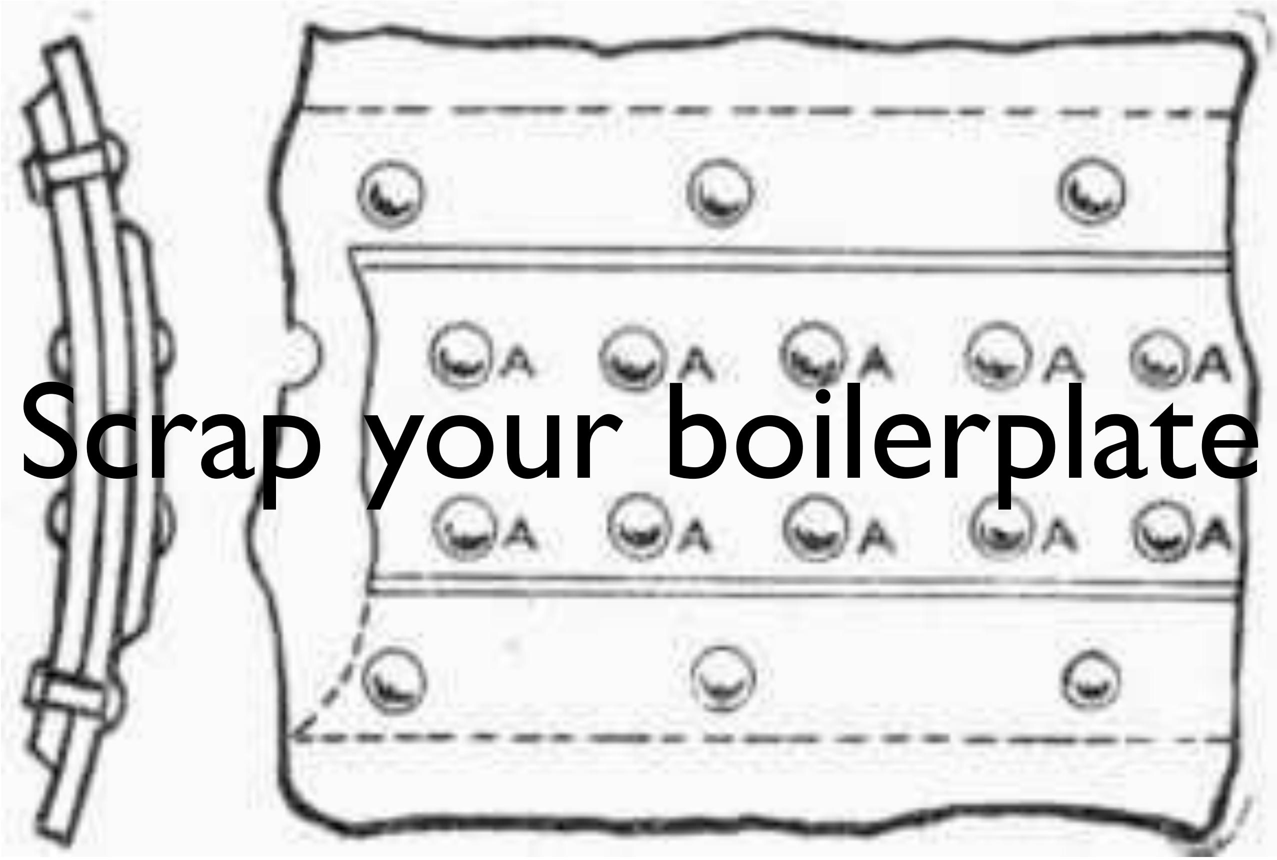
For sets, mappend is “union” of course.

Free-variable analysis based on algebra customization

```
fv :: Term -> Set Name
fv = foldTerm fvAlgebra


fvAlgebra :: TermAlgebra (Set Name)
fvAlgebra = malgebra {
  var = singleton,
  lambda = delete
}
```

Exercise: Design an algebra for mapping (as opposed to reduction) so that terms are reconstructed by default. Implement a tiny program transformation to use the new fold algebra for *Term*.

Thanks to Simon Peyton Jones for joint work.



Scrap your boilerplate

http://chestofbooks.com/crafts/metal/Applied-Science-Metal-Workers/348-Thickness-Of-Boiler-Plate.html

The code from this part is separately available through the 101companies corpus:
http://sourceforge.net/apps/mediawiki/developers/index.php?title=101companies

Totaling salaries in a company

```
sampleCompany = ( "meganalysis"
 , [ Dept "Research"
      (Employee "Craig" "Redmond" 123456)
      [ PU (Employee "Erik" "Utrecht" 12345)
      , PU (Employee "Ralf" "Koblenz" 1234)
      ]
   , Dept "Development"
      (Employee "Ray" "Redmond" 234567)
       [ DU (Dept "Dev1"
             (Employee "Klaus" "Boston" 23456)
             [ DU (Dept "Dev1.1"
                   (Employee "Karl" "Riga" 2345)
                   [ PU (Employee "Joe" "Wifi City" 2344)
                   ])
             ])
       ]
   ]
 )
```

```
> total sampleCompany
399747.0
```

Totaling salaries in a company

```
total :: Company -> Float
total = sum . map dept . snd
 where
  dept :: Dept -> Float
  dept (Dept _ m sus)
    = sum (employee m : map subunit sus)
  employee :: Employee -> Float
  employee (Employee _ _ s) = s
  subunit :: SubUnit -> Float
  subunit (PU e) = employee e
  subunit (DU d) = dept d
```

One function per possible type

Exercise: Produce some more boilerplate code. That is, define a function that only totals manager salaries.

```
> total sampleCompany
399747.0
```

Companies

Heterogenous types as opposed to a container type constructor

```
type  Company = (Name, [Dept])
data  Dept     = Dept Name Manager [SubUnit]
type  Manager = Employee
data  Employee = Employee Name Address Salary
data  SubUnit = PU Employee | DU Dept
type  Name     = String
type  Address = String
type  Salary   = Float
```

Arbitrary nesting of departments

Structural type as opposed to nominal type

“Many” types with “many” constructors

Raising salaries in a company

```
cut :: Company -> Company
cut (n,ds) = (n,map dept ds)
 where
  dept :: Dept -> Dept
  dept (Dept n m sus)
    = Dept n (employee m) (map subunit sus)
  employee :: Employee -> Employee
  employee (Employee n a s) = Employee n a (s/2)
  subunit :: SubUnit -> SubUnit
  subunit (PU e) = PU (employee e)
  subunit (DU d) = DU (dept d)
```

SYB style generic programming

Traversal scheme to query values from all subterms

Type case to return subterms of type float and 0 for other types

```
total :: Company -> Float
total = everything (+) (mkQ 0 id)

cut :: Company -> Company
cut = everywhere (mkT (/(2::Float)))
```

Traversal scheme to transform all subterms

Type case to transform floats (divide by to) and keep terms of other types

Type cast and case

This is essentially pseudo-code.
cast could be considered primitive.

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = if type of x = b then Just x else Nothing
```

```
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
mkT f = maybe id id (cast f)
```

Try to cast *f::b->b* to *a->a*; apply *f* if cast succeeds, apply *id* otherwise

```
mkQ :: (Typeable a, Typeable b) => r -> (b -> r) -> a -> r
mkQ r f x = maybe r f (cast x)
```

Try to cast *x::a* to *b*; apply *f* to *x* if cast succeeds, return *r* otherwise

Exercise: Explain how type inference is essential for the definitions of mkT and mkQ.

Generic traversal schemes

"Generic transformations"

```
type GenericT = ∀ a. Data a => a -> a

everywhere :: GenericT -> GenericT
everywhere f = f . gmapT (everywhere f)
```

Layer-by-layer map over the term.

"Generic queries"

```
type GenericQ r = ∀ a. Data a => a -> r

everything :: (r -> r -> r) -> GenericQ r -> GenericQ r
everything k f x = foldl k (f x) (gmapQ (everything k f) x)
```

Layer-by-layer map&reduce over the term.

Pseudo-code for generic one-layer traversal

Map *f* over immediate subterms, preserve constructor *C*.

```
gmapT :: GenericT -> GenericT
gmapT f (C t1 ... tn) = C (f t1) ... (f tn)

gmapQ :: GenericQ r -> GenericQ [r]
gmapQ f (C t1 ... tn) = [(f t1), ..., (f tn)]
```

Map *f* over immediate subterms, collect results in list.

There is a very polymorphic fold operation (omitted here) which admits those maps needed above. Hence, cast and that fold are the SYB primitives.

Further reading

- "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire"
  by Erik Meijer, Maarten Fokkinga, and Ross Paterson, Proceedings of FPCA 1991.
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.125
- "A fold for all seasons"
  by Tim Sheard and Leonidas Fegaras, Proceedings of FPCA 1993.
  http://portal.acm.org/citation.cfm?id=165216
- "A tutorial on the universality and expressiveness of fold"
  by Graham Hutton, Journal of Functional Programming 1999.
  http://www.citeulike.org/user/pintman/article/468391
- "Dealing with large bananas"
  by Ralf Lämmel, Joost Visser, and Jan Kort, Proceedings of WGP 2000.
  http://homepages.cwi.nl/~ralf/wgp00/
- "Scrap your boilerplate ...." (various papers)
  by various authors since 2003.
  http://sourceforge.net/apps/mediawiki/developers/index.php?title=ScrapYourBoilerplate
- "The Essence of Data Access in Comega"
  by Gavin M. Bierman, Erik Meijer, and Wolfram Schulte, Proceeedings of ECOOP 2005.
  http://dx.doi.org/10.1007/11531142_13
- "The Essence of the Iterator Pattern"
  by Jeremy Gibbons and Bruno Oliveira, Proceedings of MSFP 2006.
  http://lambda-the-ultimate.org/node/1410
- "Google's MapReduce Programming Model -- Revisited"
  by Ralf Lämmel, Science of Computer Programming 2008.
  http://userpages.uni-koblenz.de/~laemmel/MapReduce/

Thanks!
Questions and comments welcome.