Going bananas

Ralf Lämmel
Software Language Engineer
University of Koblenz-Landau
Germany

The paper!

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:

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

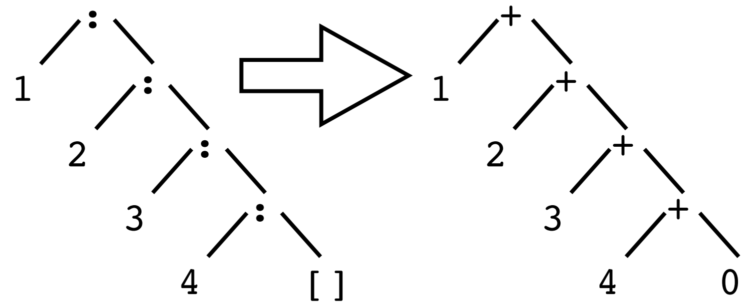
In the notation of Bird&Wadler [5] one would write $h = \mathrm{foldr} \; b \; (\oplus)$. We write catamorphisms by wrapping the relevant constituents between so called banana brackets:

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

This is why folds are called bananas.

Simple uses of **foldr**

```
> let l1 = [1,2,3,4]
> sum l1
10
```




```
> product l1
24
```

```
sum, product :: Num a => [a] -> a
sum = foldr (+) 0
product = foldr (*) 1
```

How to replace (:)

How to replace [ ]

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

Reduction with *monoids*

```
class Monoid a where
  mappend  :: a -> a -> a      -- associative operation
  mempty   :: a                -- identity of mappend
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Laws: x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
x `mappend` mempty = mempty `mappend` x = x

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

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

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

Mapping with **foldr**

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

"type-preserving"

"type-changing"

```
> map odd l1
[True,False,True,False]
```

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

Preserve list shape

Apply *f* per element

```
map f [1,2,3,4] = [f 1,f 2,f 3,f 4]
```

Mapping with *monadic effects*

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

Counter

```
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
  k = return []
  f x mys = do y <- g x; ys <- mys; return (y:ys)
```

Mapping with *applicative functors*

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

mapA :: Applicative m => (a -> m b) -> [a] -> m [b]
mapA g = foldr f k
 where
  k = pure []
  f x mys = pure (:) <*> g x <*> mys
```

Function application with effects

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

For comparison

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

More uses of **foldr**

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

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

> 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 l1
[4,3,2,1]

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

Inefficient definition

foldr vs. foldl

```
foldr f k [1,2,3,4] = 1 `f` (2 `f` (3 `f` (4 `f` k)))
foldl f k [1,2,3,4] = (((k `f` 1) `f` 2) `f` 3) `f` 4
```

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl = ... foldr ...
```

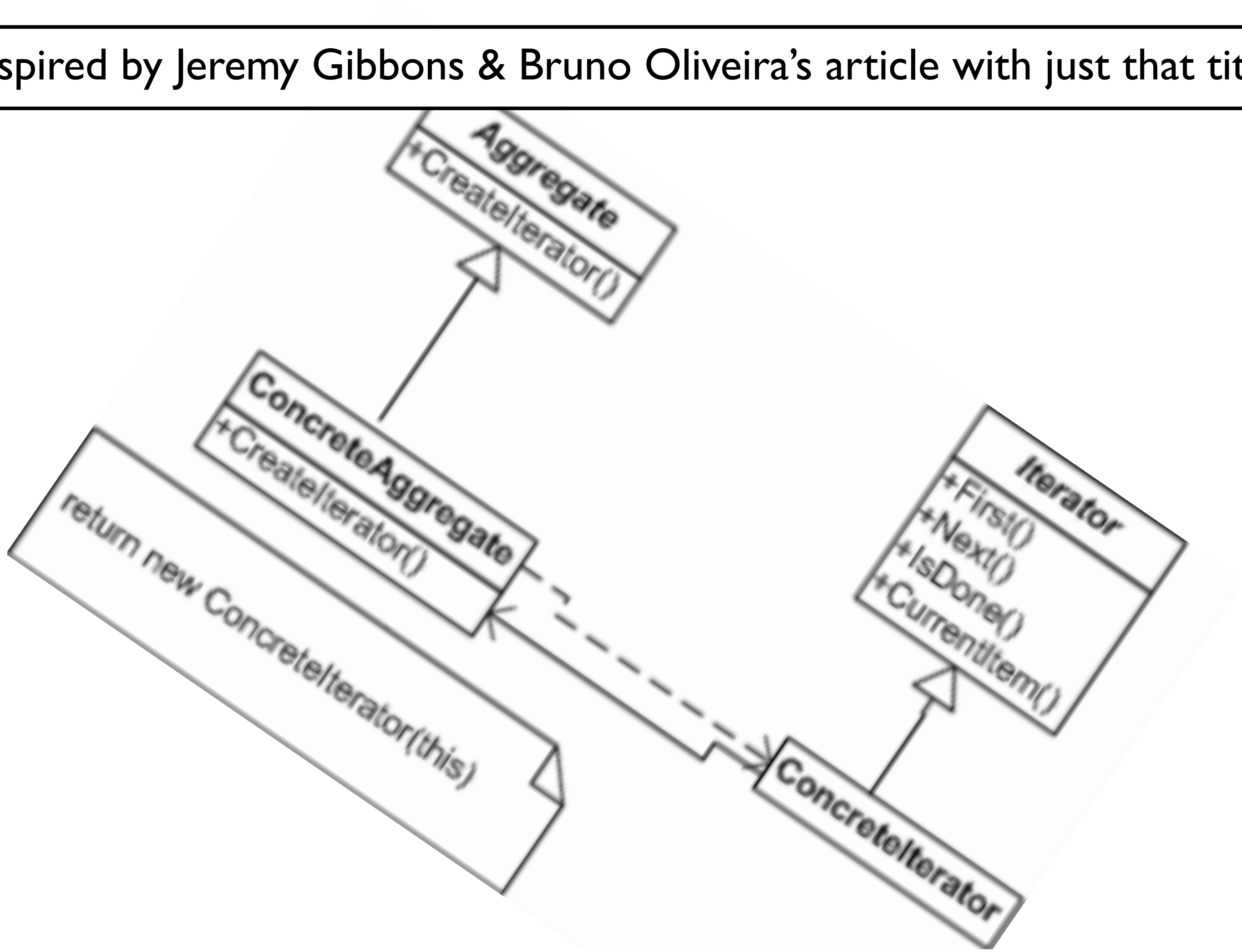
See further reading

Some million $ questions

How to ...

- ... traverse *abstract* lists?
- ... traverse lists in *parallel*?
- ... traverse *data other than lists*?

Inspired by Jeremy Gibbons & Bruno Oliveira's article with just that title.



The Essence of the Iterator Pattern

Functors: datatypes that can be *mapped*.

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

instance Functor [ ]
  where
    fmap = map
```

Exercise: How would you define *fmap* for an abstract datatype of ordered sets using search trees underneath?

Laws:

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

Foldables:
datatypes that can be *mapped* & *reduced*.

```
class Foldable t
  where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    ...

instance Foldable [ ]
  where
    foldMap f = foldr (mappend . f) mempty
    ...
```

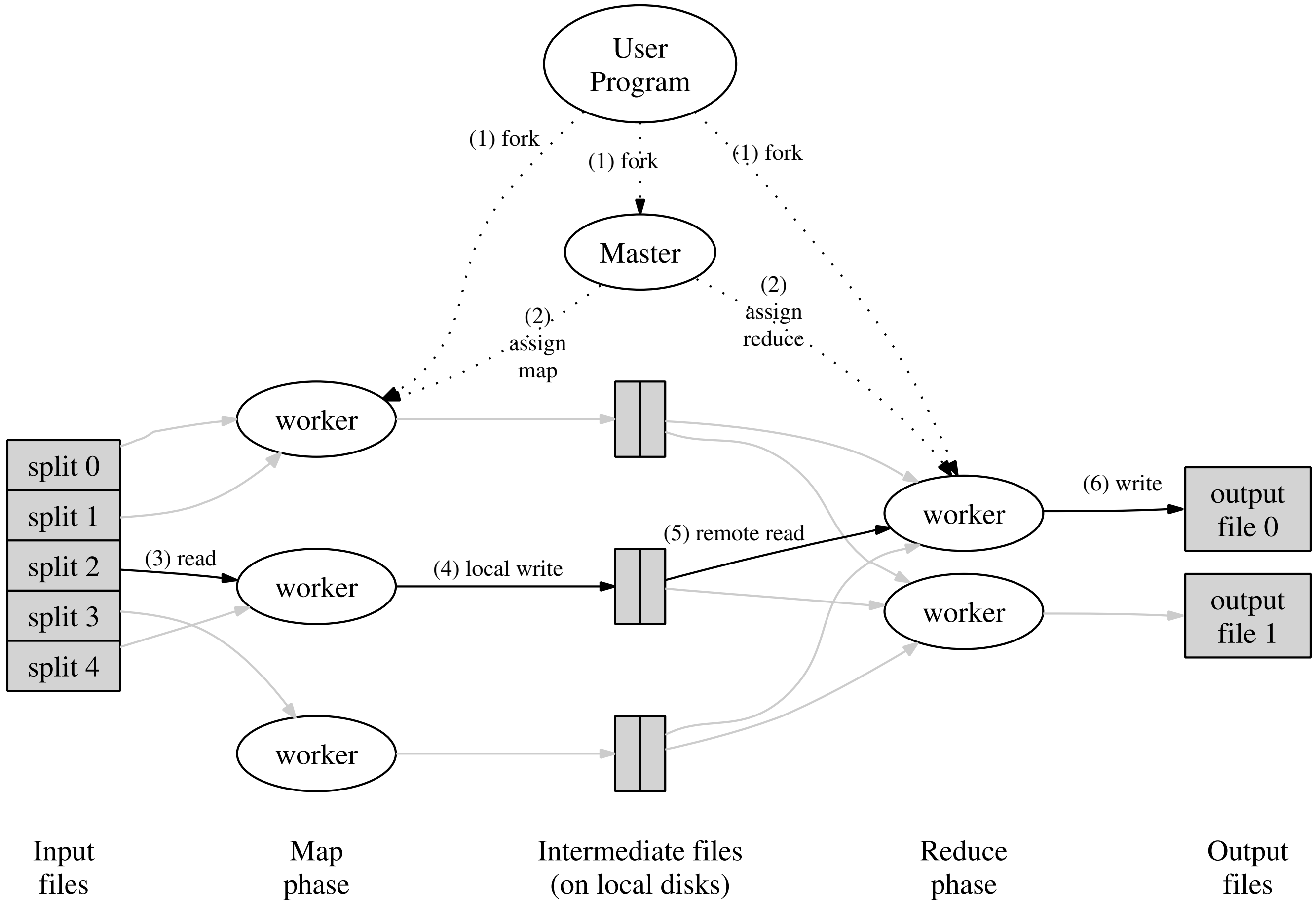
Traversables:
datatypes that can be *traversed* *with applicative functors ('effects')*

```
class (Functor t, Foldable t) => Traversable t
  where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    ...

instance Traversable [ ]
  where
    traverse = mapA
```

traverse is like a monadic map--except that applicative functors are used instead of monads.

See Ralf Lämmel's article with just that title.

Google's MapReduce Programming Model *Revisited*



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

A word index

```
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

Haskell code for the word index

```
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 = foldr (flip insert) empty
   where
    insert dict (k2,v2) = insertWith (++) k2 [v2] dict

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

Functions on Data.Map are underlined.

Exercise: Try to count the number of documents in addition to the word occurrences. Oops!

A ***parallel*** model of MapReduce

```
mapReduce'  ::  forall k1 k2 v1 v2. Ord k2
        =>  Int                                 -- Number of reducers
        ->  (k2 -> Int)                         -- Associate keys with reducers
        ->  (k1 -> v1 -> [(k2,v2)])             -- "map"
        ->  (k2 -> [v2] -> v2)                  -- "reduce"
        ->  [Map k1 v1] ->  [Map k2 v2]         -- Distributed 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" reduce

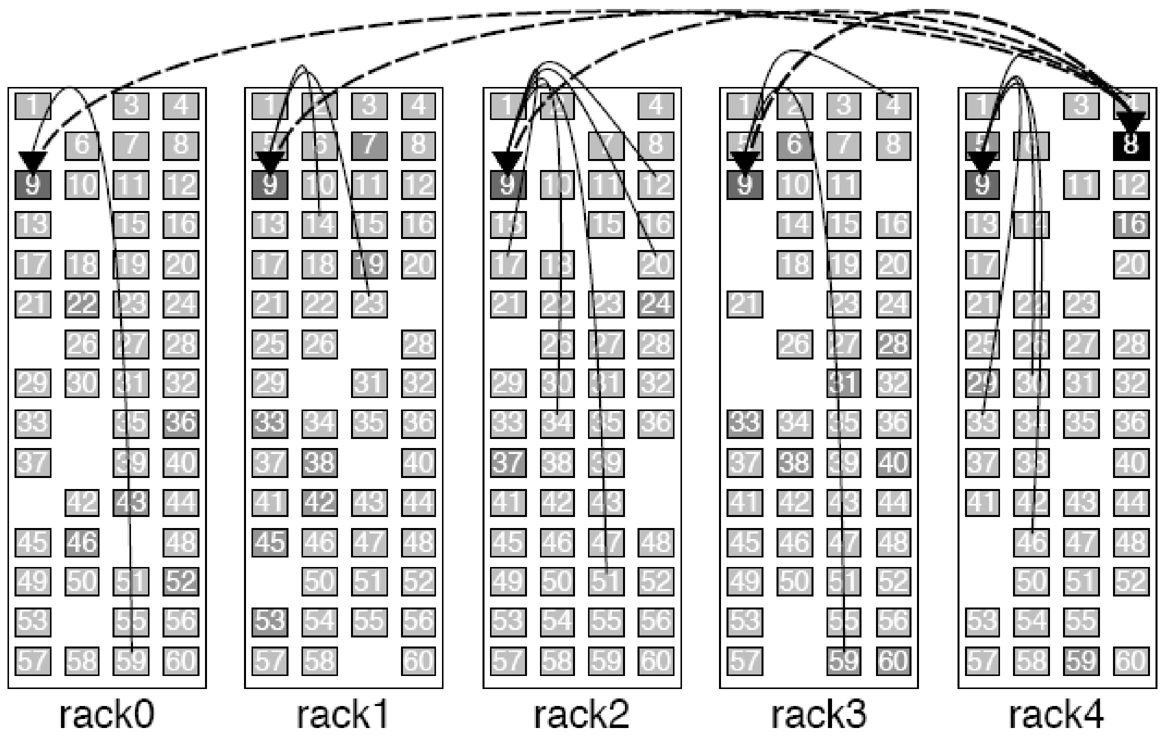
communication

"parallel" map

as before

Potentially nice, but too complicated.

Sawzall's topology
(built on top of MapReduce, really?)



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

Keys don't play a central role any longer.
Reduction is done differently in a topology sense.

List homomorphisms to the rescue

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

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

```
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)
```

Machine level

Rack level

Global level

What's the monoid for counting word occurrences?

Exercise: Generalize the shown functions so that abstract lists are used in all places.

The monoid for the word index

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

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)

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
```

The monoidal behavior of *Map* cannot be used.

Map-level append relies on value-level append.

Monoidal code for the word index

```
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: Just like in a previous exercise, return both the map for word occurrences and the document count. You need monoids on pairs.

Thanks to Joost Visser & Jan Kort for joint work.

Exercise: try to figure out how this topic relates to visitors.

Dealing with large bananas

```
data Term =
          Var Name
    |     Lambda Name Term
    |     Apply Term Term
    |     Zero
    |     Succ Term
    |     ...
```

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 = ... ...  ....
```

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

Fold algebras and folds for arbitrary (systems of) datatypes

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

One component per constructor.

Replace datatype by type parameter.

```
data Term =
          Var Name
     |    Lambda Name Term
     |    Apply Term Term
     |    Zero
     |    Succ Term
     |    ...
```

```
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)
    ...
```

Recurse and combine with the algebra

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 = ...
}
```

No recursion!
No syntax references!

Another compositional 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: the analysis predominantly combines intermediate results by union.

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''
}
```

Sets define a monoid with union as binary operation and the emtpy set as identity.

Free-variable analysis based on algebra customization

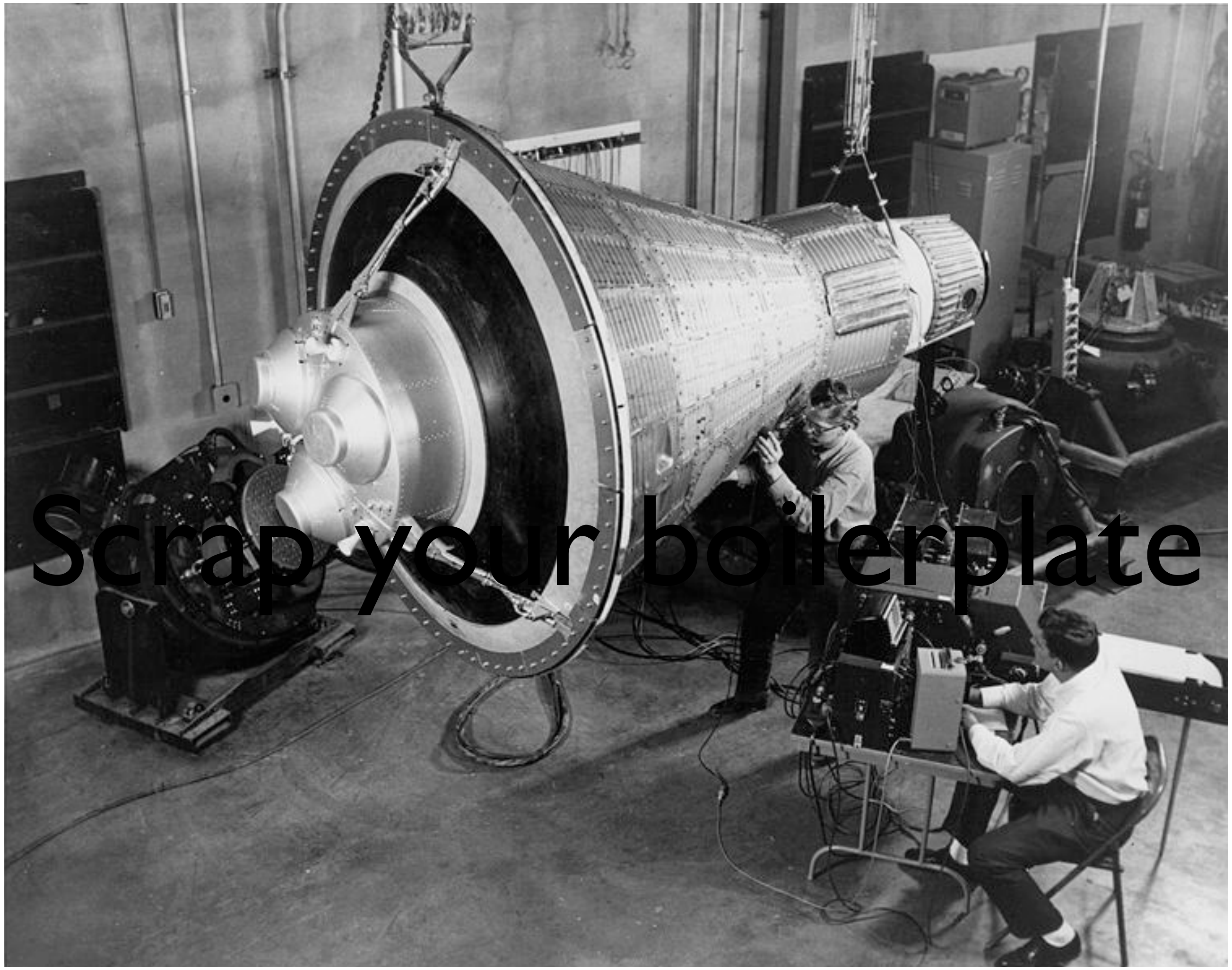
```
fv :: Term -> Set Name
fv = foldTerm fvAlgebra


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

This code is concise and clear*!*

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

Thanks to Simon Peyton Jones for joint work.

Scrap your boilerplate



http://commons.wikimedia.org/wiki/File:Boilerplate_Mercury_Capsule_-_GPN-2000-003008.jpg

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
```

Boilerplate code for total

```
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
```

1 function per type.
1 case per constructor.

Interesting code

> total sampleCompany
399747.0

Exercise: Produce more boilerplate code with a function that only totals manager salaries.

Data types for companies

Heterogenous types
as opposed to
a 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

Structural type
as opposed to
nominal type

“Many” types with
“many” constructors

~~Raising~~ Cutting 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)
```

Interesting code

SYB style generic programming

Traversal scheme to query all subterms

Override polymorphic constant function 0 to return floats when present.

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

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

Traversal scheme to transform all subterms

Override polymorphic identity to divide by 2 when faced with floats.

Generic function construction

generic default

type-specific case

polymorphic argument

```
extT g s x
 = case cast s of
      Nothing -> g x
      (Just s') -> s' x
```

cast is a primitive operation.

Likewise for extQ.

Generic traversal schemes

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

Generic transformations and queries

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

Recursively transform all children, then transform root.

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

Recursively query all children, then fold over the intermediate results using the root query for the initial value.

Pseudo-code for generic one-layer traversal

gmapT :: GenericT -> GenericT

gmapT f (C t_1 ... t_n) = C (f t_1) ... (f t_n)

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

gmapQ :: GenericQ r -> GenericQ [r]

gmapQ f (C t_1 ... t_n) = [(f t_1), ..., (f t_n)]

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

gmapT and gmapQ are defined in terms of a **gfoldl** primitive.

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.