Discrimination is Wrong

And What We Can Do About It

Edward Kmett

What?

- Discrimination is a generalization of sorting and partitioning that can be defined generically by structural recursion.

- Radix / American Flag Sort for algebraic data types.

- It breaks the classic comparison-based $\Theta(n \log n)$ bound by not working with mere pair-wise comparisons, but extracting more structure.

Why?

- **“You can do almost everything in linear time”**
- Where everything includes:
  - Sorting
  - Partitioning
  - Joining Tables
  - Constructing Maps and Sets.

Who?

- Fritz Henglein

Where?

- Director of HIPERFIT Research Center
- Professor at Univ. of Copenhagen

When?

A bunch of papers and talks from 2007-2013:

- *2011 Generic multiset programming with discrimination-based joins and symbolic Cartesian products*
- *2010 Generic Top-down Discrimination for Sorting and Partitioning in Linear Time*
- *2009 Generic Top-down Discrimination*
- *2007 Generic Discrimination Sorting and Partitioning Unshared Data in Linear Time*

Building a Nice API

“**Monads** are Monoids
in the Category of Endofunctors.

What is the Problem?”

–Stereotypical Haskell Programmer

Monoids

```
class Monoid m where
 mappend :: m -> m -> m
 mempty :: m
```

Monoidal Categories

A *monoidal category* (C,⊗,I) is a category C equipped with:

- a bifunctor (⊗) :: C * C -> C
- an object I :: C
- and natural isomorphisms
  - ρ :: (A ⊗ I) ~ A
  - λ :: (I ⊗ A) ~ A
  - α :: (A ⊗ B) ⊗ C ~ A ⊗ (B ⊗ C)

$$
\begin{tikzcd}
(A \otimes I) \otimes B \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "\rho_A \otimes 1_B"'] & & A \otimes (I \otimes B) \arrow[dl, "1_A \otimes \lambda_B"] \\
& A \otimes B &
\end{tikzcd}
$$

$$
\begin{tikzcd}
((A \otimes B) \otimes C) \otimes D \arrow[r, "\alpha_{A,B,C} \otimes 1_D"] \arrow[d, "\alpha_{A \otimes B,C,D}"'] & (A \otimes (B \otimes C)) \otimes D \arrow[r, "\alpha_{A,B \otimes C,D}"] & A \otimes ((B \otimes C) \otimes D) \arrow[d, "1_A \otimes \alpha_{B,C,D}"] \\
(A \otimes B) \otimes (C \otimes D) \arrow[rr, "\alpha_{A,B,C \otimes D}"'] & & A \otimes (B \otimes (C \otimes D))
\end{tikzcd}
$$

Products

Hask is a category with types as objects and functions as arrows.

(Hask, `(,)`, `()`) is a monoidal category with:

- ρ = `fst :: (a,()) -> a`
- ρ^{-1} = `\a -> (a, ())`
- λ = `snd :: ((), a) -> a`
- λ^{-1} = `\a -> ((), a)`
- α `:: ((a,b),c) -> (a,(b,c))`

Products and Coproducts

(Hask,`(,)`,`()`) is a monoidal category with:

- ρ = `fst :: (a, ()) -> a`
- ρ^{-1} = `\a -> (a, ())`
- λ = `snd :: ((), a) -> a`
- λ^{-1} = `\a -> ((), a)`
- α `:: ((a,b),c) -> (a,(b,c))`

(Hask,`(+)`,`Void`) is a monoidal category with:

- ρ = `(\(Left a) -> a) :: a + Void -> a`
- ρ^{-1} = `Left`
- λ = `(\(Right a) -> a) :: Void + a -> a`
- λ^{-1} = `Right`
- α `::(a + b) + c -> a + (b + c)`

Functor Composition

$\mathrm{Hask}^{\mathrm{Hask}}$ is a category with functors as objects and natural transformations as arrows.

```
type a ~> b = forall i. a i -> b i
```

($\mathrm{Hask}^{\mathrm{Hask}}$,`Compose`,`Identity`) is a monoidal category with:

- ρ `:: Compose a Identity ~> a`
- ρ `= fmap runIdentity . getCompose`
- ρ^{-1} `= Compose . fmap Identity`
- α `:: Compose (Compose a b) c ~> Compose a (Compose b c)`

Monoid Objects

A monoid object in a monoidal category (C,⊗,I) consists of

- a carrier object M
- η :: I -> M
- μ :: M ⊗ M -> M

such that:

$$
\begin{tikzcd}
I \otimes M \arrow[r, "\eta\otimes 1"] \arrow[dr, "\lambda"'] & M \otimes M \arrow[d, "\mu"] & M \otimes I \arrow[l, "1\otimes\eta"'] \arrow[dl, "\rho"] \\
& M &
\end{tikzcd}
$$

$$
\begin{tikzcd}
(M \otimes M) \otimes M \arrow[r, "\alpha"] \arrow[d, "\mu\otimes 1"'] & M \otimes (M \otimes M) \arrow[r, "1\otimes\mu"] & M \otimes M \arrow[d, "\mu"] \\
M \otimes M \arrow[rr, "\mu"'] & & M
\end{tikzcd}
$$

Monoids as Monoid Objects

A monoid object in (Hask, `(,)`, `()`) is an object M with

```
η :: () -> M
η () = mempty

μ :: (M,M) -> M
μ = uncurry mappend
```

such that the `Monoid` laws hold.

Monads as Monoid Objects

A monoid object in ($Hask^{Hask}$,`Compose`,`Identity`) is a `Functor` M with

```
η :: Identity ~> M
η = return . runIdentity
```

```
μ :: Compose M M ~> M
μ = join . getCompose
```

such that the monad laws hold.

Day Convolution from `(<*>)`

```
data Day f g a where
  Day :: f (a -> b) -> g a -> Day f g b

(<*>) :: Applicative f => f (a -> b) -> f a -> f b

Day (<*>) :: Day f f ~> f
```

Applicatives as Monoid Objects

A monoid object in ($\mathrm{Hask}^{\mathrm{Hask}}$,`Day`,`Identity`) is a `Functor` M with

```
η :: Identity ~> M
η = pure . runIdentity

μ :: Day M M ~> M
μ (Day m n) = m <*> n
```

such that the Applicative laws hold on M.

“**Applicatives** are Monoids
in the Category of Endofunctors.

What is the Problem?”

–Me

Day Convolution from `liftA2`

Covariant Day Convolution:

```
data Day f g a where
  Day :: ((a ⊗₁ b) -> c) ⊗₂ f a ⊗₂ g b -> Day f g c
```

Contravariant Day Convolution:

```
data Day f g a where
  Day :: (c -> (a ⊗₁ b)) ⊗₂ f a ⊗₂ g b -> Day f g c
```

Is There a Contravariant _______?

Is There a Contravariant __Monad__?

No

Is There a Contravariant Comonad?

No

Is There a Contravariant Applicative?

Yes!

Divide and Conquer

```
class Contravariant f => Divisible f where
 divide  :: (a -> (b, c)) -> f b -> f c -> f a
 conquer :: f a
```

comes from contravariant Day Convolution:

```
data Day f g a where
  Day :: (c -> (a ⊗₁ b)) ⊗₂ f a ⊗₂ g b -> Day f g c
```

with

```
⊗₁ = (,)
⊗₂ = (,)
```

Divide and Conquer w/ Some Laws

```
class Contravariant f => Divisible f where
 divide  :: (a -> (b, c)) -> f b -> f c -> f a
 conquer :: f a

delta a = (a,a)
divide delta m conquer = m
divide delta conquer m = m
divide delta (divide delta m n) o = divide delta m
(divide delta n o)
```

Divide and Conquer w/ Real Laws

```
class Contravariant f => Divisible f where
 divide  :: (a -> (b, c)) -> f b -> f c -> f a
 conquer :: f a

divide f m conquer = contramap (fst . f) m
divide f conquer m = contramap (snd . f) m
divide f (divide g m n) o
  = divide f' m (divide id n o)
  where
    f' a = case f a of
      (bc,d) -> case g bc of
        (b,c) -> (a,(b,c))
```

Choose and Lose

```
class Divisible f => Decidable f where
  choose :: (a -> Either b c) -> f b -> f c -> f a
  lose :: (a -> Void) -> f a
```

comes from contravariant Day Convolution:

```
data Day f g a where
  Day :: (c -> (a ⊗₁ b)) ⊗₂ f a ⊗₂ g b -> Day f g c
```

with

```
⊗₁ = Either
⊗₂ = (,)
```

* The superclass constraint comes from Hask being a distributive category.

Choose and Lose

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a

class Contravariant f => Divisible f where
  divide  :: (a -> (b, c)) -> f b -> f c -> f a
  conquer :: f a

class Divisible f => Decidable f where
  choose :: (a -> Either b c) -> f b -> f c -> f a
  lose :: (a -> Void) -> f a
```

Why is there an argument to `lose`?

```
pureish     :: Applicative f => (() -> a) -> f a
emptyish    :: Alternative f => (Void -> a) -> f a
conquerish  :: Divisible f => (a -> ()) -> f a
lose        :: Decidable f => (a -> Void) -> f a

pure a = pureish (const a)
empty = emptyish absurd
conquer = conquerish (const ())
```

Predicates

```
newtype Predicate a = Predicate { getPredicate :: a -> Bool }

instance Contravariant Predicate where
  contramap f (Predicate g) = Predicate (g . f)

instance Divisible Predicate where
  divide f (Predicate g) (Predicate h) = Predicate $
    \a -> case f a of
      (b, c) -> g b && h c
  conquer = Predicate $ const True

instance Decidable Predicate where
  lose f = Predicate $ \a -> absurd (f a)
  choose f (Predicate g) (Predicate h) = Predicate $
    either g h . f
```

Op

```
newtype Op r a = Op { getOp :: a -> r }

instance Contravariant (Op r) where
  contramap f (Op g) = Op (g . f)

instance Monoid r => Divisible (Op r) where
  divide f (Op g) (Op h) = Op $
    \a -> case f a of
      (b, c) -> g b <> h c
  conquer = Op $ const mempty

instance Monoid r => Decidable (Op r) where
  lose f = Op $ \a -> absurd (f a)
  choose f (Op g) (Op h) = Op $
    either g h . f
```

Equivalence Classes

```
newtype Equivalence a = Equivalence { getEquivalence :: a -> a -> Bool }

instance Contravariant Equivalence where
  contramap f g = Equivalence $ on (getEquivalence g) f

instance Divisible Equivalence where
  divide f (Equivalence g) (Equivalence h) = Equivalence $ \a b -> case f a of
    (a',a'') -> case f b of
      (b',b'') -> g a' b' && h a'' b''
  conquer = Equivalence $ \_ _ -> True

instance Decidable Equivalence where
  lose f = Equivalence $ \a -> absurd (f a)
  choose f (Equivalence g) (Equivalence h) = Equivalence $ \a b -> case f a of
    Left c -> case f b of
      Left d -> g c d
      Right{} -> False
    Right c -> case f b of
      Left{} -> False
      Right d -> h c d
```

Comparisons

```
newtype Comparison a = Comparison { getComparison :: a -> a -> Ordering }

instance Contravariant Comparison where
  contramap f g = Comparison $ on (getComparison g) f

instance Divisible Comparison where
  divide f (Comparison g) (Comparison h) = Comparison $ \a b -> case f a of
    (a',a'') -> case f b of
      (b',b'') -> g a' b' <> h a'' b''
  conquer = Comparison $ \_ _ -> EQ

instance Decidable Comparison where
  lose f = Comparison $ \a _ -> absurd (f a)
  choose f (Comparison g) (Comparison h) = Comparison $ \a b -> case f a of
    Left c -> case f b of
      Left d -> g c d
      Right{} -> LT
    Right c -> case f b of
      Left{} -> GT
      Right d -> h c d
```

Deciding with Generics

```
class GDeciding q t where
 gdeciding :: Decidable f => p q -> (forall b. q b => f b) -> f (t a)

instance (GDeciding q f, GDeciding q g) => GDeciding q (f :*: g) where
  gdeciding p q = divide (\(a :*: b) -> (a, b))
    (gdeciding p q) (gdeciding p q)

instance GDeciding q U1 where
  gdeciding _ _ = conquer

instance (GDeciding q f, GDeciding q g) => GDeciding q (f :+: g) where
  gdeciding p q = choose (\ xs -> case xs of L1 a -> Left a; R1 a -> Right a)
    (gdeciding p q) (gdeciding p q)

instance GDeciding q V1 where
  gdeciding _ _ = lose (\ !_ -> error “impossible")
```

Using Generic Decisions

```
deciding :: (Deciding q a, Decidable f)
  => p q -> (forall b. q b => f b) -> f a
deciding p q = contramap from $ gdeciding p q

gcompare :: Deciding Ord a => a -> a -> Ordering
gcompare = getComparison $ deciding
  (Proxy :: Proxy Ord) (Comparison compare)

geq :: Deciding Eq a => a -> a -> Bool
geq = getEquivalence $ deciding
  (Proxy :: Proxy Eq) (Equivalence (==))
```

Stable **Ordered** Discrimination

Initial Encoding

```
data Order t where
 NatO :: Int → Order Int
 TrivO :: Order t
 SumL :: Order t1 → Order t2 → Order (Either t1 t2)
 ProdL :: Order t1 → Order t2 → Order (t1, t2)
 MapO :: (t1 → t2) → Order t2 → Order t1
 ListL :: Order t → Order [t]
 BagO :: Order t → Order [t]
 SetO :: Order t → Order [t]
 Inv :: Order t → Order t
```

Final Encoding

```
newtype Sort a = Sort
  { runSort :: forall b. [(a,b)] -> [[b]]
  }
```

Final Encoding (w/ Instances)

```
newtype Sort a = Sort
  { runSort :: forall b. [(a,b)] -> [[b]]
  }

instance Contravariant Sort where
  contramap f (Sort g) = Sort $ g . fmap (first f)

instance Divisible Sort where …

instance Decidable Sort where …
```

Sorting with Class

```
class Sorting a where
  sorting :: Sort a
  default sorting :: Deciding Sorting a => Sort a
  sorting = deciding (Proxy :: Proxy Sorting) sorting

instance Sorting Void
instance Sorting Bool
instance Sorting a => Sorting [a]
instance Sorting a => Sorting (Maybe a)
instance (Sorting a, Sorting b) => Sorting (Either a b)
instance (Sorting a, Sorting b) => Sorting (a, b)
instance (Sorting a, Sorting b, Sorting c) => Sorting (a, b, c)
instance (Sorting a, Sorting b, Sorting c, Sorting d) => Sorting (a, b, c, d)
…
```

Sorting Law

For any **strictly monotone-increasing** function `f`

```
contramap f sorting = sorting
```

Divisible Sort

```
newtype Sort a = Sort
  { runSort :: forall b. [(a,b)] -> [[b]]
  }

instance Divisible Sort where
  conquer = Sort $ return . fmap snd

  divide k (Sort l) (Sort r) = Sort $ \xs ->
    l [ (b, (c, d)) | (a,d) <- xs, let (b, c) = k a]
    >>= r
```

Decidable Sort

```
newtype Sort a = Sort
  { runSort :: forall b. [(a,b)] -> [[b]]
  }

instance Decidable Sort where
  lose k = Sort $ fmap (absurd.k.fst)
  choose f (Sort l) (Sort r) = Sort $ \xs -> let
      ys = fmap (first f) xs
    in l [ (k,v) | (Left k, v) <- ys]
    ++ r [ (k,v) | (Right k, v) <- ys]
```

Other Base Cases

```
— Sort integers in the range [0…n-1]
sortingNat :: Int -> Sort Int

instance Sorting Word8 where
  sorting = contramap fromIntegral (sortingNat 256)

instance Sorting Word16 where
  sorting = contramap fromIntegral (sortingNat 65536)
```

American Flag Sort

```
— American Flag Sort
instance Sorting Word32 where
  sorting = divide
    ( \x -> (fromIntegral x .&. 0xffff
           , fromIntegral (unsafeShiftR x 16)
           )
    ) (sortingNat 65536) (sortingNat 65536)
```

Radix vs. the American Flag

```
— American Flag
instance Sorting Word32 where
  sorting = divide
    ( \x -> (fromIntegral x .&. 0xffff
           , fromIntegral (unsafeShiftR x 16)
           )
    ) (sortingNat 65536) (sortingNat 65536)

— Radix Sort
instance Sorting Word32 where
  sorting = Sort (runs <=< runSort (sortingNat 65536)
                 . join . runSort (sortingNat 65536)
                 . fmap radices) where
    radices (x,b) =
       (fromIntegral x .&. 0xffff,
       (fromIntegral (unsafeShiftR x 16),
       (x,b)))
```

Sorting

```
— O(n) sort for ADTs
sort :: Sorting a => [a] -> [a]
sort as = List.concat $
  runSort sorting [ (a, a) | a <- as ]

— O(n) sort with a Schwartzian transform
sortWith :: Sorting b => (a -> b) -> [a] -> [a]
sortWith f as = List.concat $
  runSort sorting [ (f a, a) | a <- as ]
```

Map/IntMap/Set/IntSet Construction

```
— O(n) Map construction
toMap :: Sorting k => [(k, v)] -> Map k v
toMap kvs = Map.fromDistinctAscList $
  last <$> runSort sorting
    [ (fst kv, kv) | kv <- kvs ]

toMapWith :: Sorting k => (v -> v -> v) -> [(k, v)]
-> Map k v
```

Result

- *O(n)* stable, ordered, structural discrimination for any ADT.
- Using radix sort instead of the American Flag sort for integers or other simple products with O(1) comparisons provides a big speed boost.
- GHC Generics let users derive the instance for their type with one line.

Stable **Unordered** Discrimination

Initial Encoding

```
data Equiv t where
  NatE :: Int → Equiv Int
  TrivE :: Equiv t
  SumE :: Equiv t1 → Equiv t2 → Equiv (Either t1 t2)
  ProdE :: Equiv t1 → Equiv t2 → Equiv (t1, t2)
  MapE :: (t1 → t2) → Equiv t2 → Equiv t1
  ListE :: Equiv t → Equiv [t]
  BagE :: Equiv t → Equiv [t]
  SetE :: Equiv t → Equiv [t]
```

Final Encoding

```
newtype Group a = Group
  { runGroup :: forall b. [(a,b)] -> [[b]]
  }
```

Why Unordered?

- `instance Eq IORef` exists, `instance Ord IORef` does not.
- `sorting` is provably unproductive, but `grouping` *could* be productive!

unsafePerformIO Inception

```haskell
groupingNat :: Int -> Group Int
groupingNat n = unsafePerformIO $ do
    ts <- newIORef ([] :: [MVector RealWorld [Any]])
    return $ Group $ go ts
  where
    step1 t keys (k, v) = read t k >>= \vs -> case vs of
      [] -> (k:keys) <$ write t k [v]
      _  -> keys     <$ write t k (v:vs)
    step2 t vss k = do
      es <- read t k
      (reverse es : vss) <$ write t k []
    go ts xs = unsafePerformIO $ do
      mt <- atomicModifyIORef ts $ \case
        (y:ys) -> (ys, Just y)
        []     -> ([], Nothing)
      t <- maybe (replicate n []) (return . unsafeCoerce) mt
      ys <- foldM (step1 t) [] xs
      zs <- foldM (step2 t) [] ys
      atomicModifyIORef ts $ \ws -> (unsafeCoerce t:ws, ())
      return zs
    {-# NOINLINE go #-}
{-# NOINLINE groupingNat #-}
```

Desired Grouping Law

For any **injective** function `f`

```
contramap f grouping = grouping
```

Why This Law?

$O(n)$ nub

```
group :: Grouping a => [a] -> [[a]]
group as = runGroup grouping [(a, a) | a <- as]

groupWith :: Grouping b => (a -> b) -> [a] -> [[a]]
groupWith f as = runGroup grouping [(f a, a) | a <- as]

nub :: Grouping a => [a] -> [a]
nub as = head <$> group as

nubWith :: Grouping b => (a -> b) -> [a] -> [a]
nubWith f as = head <$> groupWith f as
```

Potential For Streaming

```
runGroup grouping [(1,’a’),(2,’b’),(1,’c’),(3,’d’)]
    = [“ac”,”b”,”d”]
```

Trouble with the Law

What Is Wrong With Discrimination?

```
disc :: Equiv k → [(k,v)] -> [[v]]
…
disc (SumE e1 e2) xs =
  disc e1 [ (k, v) | (Left k, v) <- xs ] ++
  disc e2 [ (k, v) | (Right k, v) <- xs ]

disc (ProdE e1 e2) xs =
  [ vs | ys <- disc e1 [ (k1, (k2, v))
                       | ((k1, k2), v) ← xs
                       ]
       , vs <- disc e2 ys
  ]
…
```

* from *Generic Top-down Discrimination for Sorting and Partitioning in Linear Time*

An Unproductive Fix

```
legal :: Group a -> Group a
legal (Group g) = Group $ \xs -> do
 zs <- g $ zipWith (\n (a,d) -> (a, (n, d))) [0..] xs
  fmap snd <$> sortWith (\((n,d):_) -> n) zs
```

Fixing Sums Productively

```
choose f (Group l) (Group r) = Group $ \xs -> let
      ys = zipWith (\n (a,d) -> (f a, (n, d))) [0..] xs
    in l [ (k,p) | (Left k, p) <- ys ] `mix`
       r [ (k,p) | (Right k, p) <- ys ]

mix :: [[(Int,b)]] -> [[(Int,b)]] -> [[b]]
mix [] bs = fmap snd <$> bs
mix as [] = fmap snd <$> as
mix asss@(((n,a):as):ass) bsss@(((m,b):bs):bss)
  | n < m     = (a:fmap snd as) : mix ass bsss
  | otherwise = (b:fmap snd bs) : mix asss bss
mix _ _ = error "bad discriminator"
```

Discriminating Pointers

Grouping STRefs in O(n)

```
foreign import prim "walk" walk :: Any -> MutableByteArray# s -> State# s -> (# State# s, Int# #)

groupingSTRef :: Group Addr -> Group (STRef s a)
groupingSTRef (Group f) = Group $ \xs ->
  let force !n !(!(STRef !_,_):ys) = force (n + 1) ys
      force !n [] = n
  in case force 0 xs of
   !n -> unsafePerformIO $ do
     mv@(PM.MVector _ _ (MutableByteArray mba)) <- PM.new n :: IO (PM.MVector RealWorld Addr)
     IO $ \s -> case walk (unsafeCoerce xs) mba s of (# s', _ #) -> (# s', () #)
     ys <- P.freeze mv
     return $ f [ (a,snd kv) | kv <- xs | a <- P.toList ys ]
{-# NOINLINE groupingSTRef #-}
```

Adding a foreign prim

```
#include "Cmm.h"


walk(P_ lpr, P_ mba)
{
  W_ i;
  i = 0;

  W_ list_clos;
  list_clos = UNTAG(lpr);

walkList:
  W_ type;
  type = TO_W_(%INFO_TYPE(%GET_STD_INFO(list_clos)));

  switch [INVALID_OBJECT .. N_CLOSURE_TYPES] type {
    case IND, IND_PERM, IND_STATIC: { /* indirection */
      list_clos = UNTAG(StgInd_indirectee(list_clos));
      goto walkList; /* follow it and try again */
    }
    case CONSTR_STATIC: { /* [] */
      goto walkNil;
    }
    case CONSTR_2_0: { /* pair_clos:next_clos */
       P_ pair_clos, next_clos;
       pair_clos = UNTAG(StgClosure_payload(list_clos, 0));
       next_clos = UNTAG(StgClosure_payload(list_clos, 1));
walkPair:
       // .. process the pair
       type = TO_W_(%INFO_TYPE(%GET_STD_INFO(pair_clos)));
       switch [INVALID_OBJECT .. N_CLOSURE_TYPES] type {
         case IND, IND_PERM, IND_STATIC: { /* indirection */
           pair_clos = UNTAG(StgInd_indirectee(pair_clos));
           goto walkPair; /* follow it and try again */
         }
         case CONSTR_2_0: { /* (r,a) */
           P_ ioref_clos;
           ioref_clos = UNTAG(StgClosure_payload(pair_clos, 0)); // fst
walkIORef:
           type = TO_W_(%INFO_TYPE(%GET_STD_INFO(ioref_clos)));
           switch [INVALID_OBJECT .. N_CLOSURE_TYPES] type {
             case IND, IND_PERM, IND_STATIC: {
               ioref_clos = UNTAG(StgInd_indirectee(ioref_clos));
               goto walkIORef;
             }
             case CONSTR_1_0: {
               P_ mutvar_clos;
               mutvar_clos = UNTAG(StgClosure_payload(ioref_clos, 0)); // retrieve the MutVar#
walkMutVar:
               type = TO_W_(%INFO_TYPE(%GET_STD_INFO(mutvar_clos)));
               switch [INVALID_OBJECT .. N_CLOSURE_TYPES] type {
                 case IND, IND_PERM, IND_STATIC: {
                   mutvar_clos = UNTAG(StgInd_indirectee(mutvar_clos));
                   goto walkMutVar;
                 }
                 case MUT_VAR_CLEAN, MUT_VAR_DIRTY: {
                   W_[mba + i] = TO_W_(mutvar_clos);
                   i = i + 1;
                   list_clos = next_clos;
                   goto walkList;
                 }
                 default: {
                   ccall barf("walk: unexpected MutVar# closure type entered!") never returns;
                 }
               }
             }
             default: {
               ccall barf("walk: unexpected IORef closure type entered!") never returns;
             }
           }
         }
         default: {
           ccall barf("walk: unexpected product closure type entered!") never returns;
         }
       }
    }
    default: {
      ccall barf("walk: unexpected list closure type entered!") never returns;
    }
  }
walkNil:
  return (0);
```

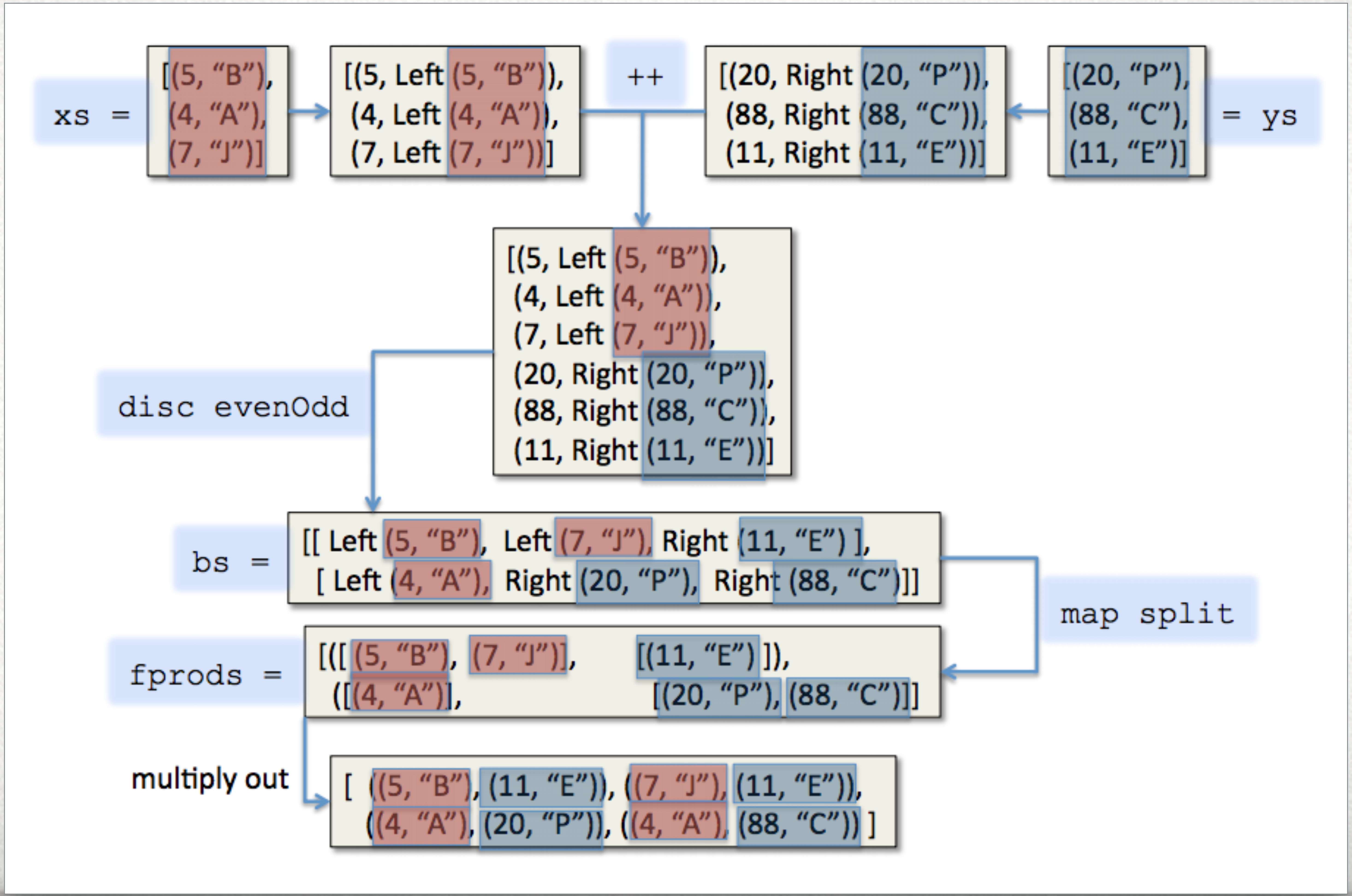
Indiscriminate Discrimination

```
class Decidable f => Discriminating f where
  disc :: f a -> [(a, b)] -> [[b]]

instance Discriminating Sort where
  disc (Sort f) = f

instance Discriminating Group where
  disc (Group g) = g
```

Joins



xs =
[(5, "B"),
(4, "A"),
(7, "J")]
[(5, Left (5, "B")),
(4, Left (4, "A")),
(7, Left (7, "J"))]
++
[(20, Right (20, "P")),
(88, Right (88, "C")),
(11, Right (11, "E"))]
[(20, "P"),
(88, "C"),
(11, "E")]
= ys
[(5, Left (5, "B")),
(4, Left (4, "A")),
(7, Left (7, "J")),
(20, Right (20, "P")),
(88, Right (88, "C")),
(11, Right (11, "E"))]
disc evenOdd
bs =
[[Left (5, "B"), Left (7, "J"), Right (11, "E")],
[Left (4, "A"), Right (20, "P"), Right (88, "C")]]
map split
fprods =
[(([(5, "B"), (7, "J")], [(11, "E")]),
([(4, "A")], [(20, "P"), (88, "C")]]
multiply out
[((5, "B"), (11, "E")), ((7, "J"), (11, "E")),
((4, "A"), (20, "P")), ((4, "A"), (88, "C"))]

Towards Joins

– All lefts are known to come before all rights

```
spanEither :: ([a] -> [b] -> c) -> [Either a b] -> c
spanEither k xs0 = go [] xs0 where
  go acc (Left x:xs) = go (x:acc) xs
  go acc rights = k (reverse acc) (fromRight <$> rights)
```

Outer Joins

```
outer
  :: Discriminating f
  => f d                      -- ^ the discriminator to use
  -> ([a] -> [b] -> c)  -- ^ how to join two tables
  -> (a -> d)                 -- ^ selector for the left table
  -> (b -> d)                 -- ^ selector for the right table
  -> [a]                      -- ^ left table
  -> [b]                      -- ^ right table
  -> [c]
outer m abc ad bd as bs = spanEither abc <$>
  disc m (((ad &&& Left) <$> as) ++ ((bd &&& Right) <$> bs))
```

Inner Joins

```haskell
inner
  :: Discriminating f
  => f d              -- ^ the discriminator to use
  -> (a -> b -> c)  -- ^ how to join two rows
  -> (a -> d)       -- ^ selector for the left table
  -> (b -> d)       -- ^ selector for the right table
  -> [a]            -- ^ left table
  -> [b]            -- ^ right table
  -> [[c]]
inner m abc ad bd as bs = catMaybes $ joining m go ad bd as bs where
  go ap bp
    | Prelude.null ap || Prelude.null bp = Nothing
    | otherwise = Just (liftA2 abc ap bp)
```

Open Problems

Productive Stable Unordered Discrimination

- Needs better versions of `groupingNat`, `divide`
- Likely needs a different encoding.
- Experiments with an lazy ST-like calculation that can produce lazily driven IVars look promising.

Conclusion

Discrimination gives us *O(n)*

- **`sort`**
- **`nub`**
- **`group`**
- inner/outer/left outer/right outer joins

for a very wide array of data types:

- ADTs
- integers
- pointers

Any Questions?

Code: http://github.com/ekmett/discrimination
Documentation: http://ekmett.github.io/discrimination/

There is still room for improvement:

I want **productive** unordered discrimination.

Help me get there.