Generic traversals

Roman Cheplyaka

Foldable tuples

```
> length (3,4)
```

Foldable tuples

Michael Snoyman
@snoyberg

Following

Days since last mailing list discussion of Foldable tuples:

0

RESTART THE CLOCK!

RETWEETS 2 LIKES 28

2:11 PM - 23 Apr 2017

Foldable tuples

```
> fmap show [1..5]
["1","2","3","4","5"]

> fmap show (3,4)
(3,"4")
```

Foldable tuples

```
> fmap show [1..5]
["1","2","3","4","5"]

> fmap show (3,4)
(3,"4")
```

From tuples-homogenous-h98:

```
> import Data.Tuple.Homogenous
> length (Tuple2 (3,4))
2
> fmap show (Tuple2 (3,4))
Tuple2 {untuple2 = ("3","4")}
```

Homogenous tuples

```
let
  temp_high_F = to_fahrenheit temp_high_C
  temp_low_F  = to_fahrenheit temp_low_C

let
  [temp_high_F, temp_low_F] =
    map to_fahrenheit [temp_high_C, temp_low_C]

let
  Tuple2 (temp_high_F, temp_low_F) =
    fmap to_fahrenheit (Tuple2 (temp_high_C, temp_low_C))
```

Heterogeneous length

```
class Lengthy a where
  length :: a -> Int

instance Lengthy (a, b) where
  length = 2
```

Heterogeneous length

```
import Data.Data
import Data.Functor.Const

length :: Data a => a -> Int
length =
  getConst .
  gfoldl (\(Const c) _ -> Const (c+1)) (const 0)

> length (3,4)
2

> length [1..10]
```

Understanding gfoldl

```haskell
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a
```

«Trying to understand the type of gfoldl directly can lead to brain damage. It is easier to see what the instances look like.»
— Ralf Lämmel & Simon Peyton Jones

Understanding gfoldl

```
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

newtype Const a b = Const { getConst :: a }

length :: Data a => a -> Int
length =
  getConst .
  gfoldl (\(Const c) _ -> Const (c+1)) (const 0)
```

Understanding gfoldl

```haskell
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

instance Data a => Data [a] where
  gfoldl f z = \case
    []   -> z []
    x:xs -> z (:) `f` x `f` xs
```

Fixing gfoldl

```haskell
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

instance Data a => Data [a] where
  gfoldl f z = \case
    []   -> z []
    x:xs -> z (:) `f` x `f` (gfoldl f z xs)
```

Fixing gfoldl

```
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

instance Data a => Data [a] where
  gfoldl f z = \case
    [] -> z []
    [x1] -> z (\x1 -> [x1]) `f` x1
    [x1,x2] -> z (\x1 x2 -> [x1, x2]) `f` x1 `f` x2
    [x1,x2,x3] -> z (\x1 x2 x3 -> [x1, x2, x3])
      `f` x1 `f` x2 `f` x3
```

Arriving at gtraverse

Understanding gfoldl

```
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

instance Data a => Data [a] where
  gfoldl f pure = \case
    []   -> pure []
    x:xs -> pure (:) `f` x `f` xs

instance Traversable [a] where
  traverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> traverse g xs
```

Understanding gfoldl

```haskell
class Data a where
  gfoldl
    :: (forall d. Data d => d -> c d)
    -> (forall d b. c (d -> b) -> c d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

instance Data a => Data [a] where
  gfoldl g (<*>) pure = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> g xs

instance Traversable [a] where
  traverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> traverse g xs
```

Understanding gfoldl

```
class Data a where
  gtraverse
    :: Applicative c
    => (forall d . Data d => d -> c d)
    -> a -> c a

instance Data a => Data [a] where
  gtraverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> g xs

instance Traversable [a] where
  traverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> traverse g xs
```

Fixing gfoldl

```
class Data a where
  gtraverse
    :: Applicative c
    => (forall d . Data d => d -> c d)
    -> a -> c a

instance Data a => Data [a] where
  gtraverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> gtraverse g xs

instance Traversable [a] where
  traverse g = \case
    []   -> pure []
    x:xs -> pure (:) <*> g x <*> traverse g xs
```

Relationship between gtraverse and gfoldl

gtraverse from gfoldl

```haskell
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

  gtraverse
    :: Applicative c
    => (forall d . Data d => d -> c d)
    -> a -> c a

  gtraverse f = gfoldl g pure
    where
      g acc x = acc <*> f x
```

gfoldl from gtraverse

```haskell
class Data a where
  gfoldl
    :: (forall d b. Data d => c (d -> b) -> d -> c b)
    -> (forall g. g -> c g)
    -> a -> c a

  gtraverse
    :: Applicative c
    => (forall d . Data d => d -> c d)
    -> a -> c a

  gfoldl f z = _ -- ???
```

go one level down

$$\frac{\text{gfoldl}}{\text{gtraverse}} = \frac{\text{foldl}}{?}$$

go one level down

$$\frac{\text{gfoldl}}{\text{gtraverse}} = \frac{\text{foldl}}{\text{foldMap}}$$

go one level down

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

  foldl :: (b -> a -> b) -> b -> t a -> b
```

go one level down

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

  foldl :: (b -> a -> b) -> b -> t a -> b

  foldl f z t =
    appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
```

go one level down

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

  foldl :: (b -> a -> b) -> b -> t a -> b

  foldl f z =
    List.foldl f z . foldMap (\x -> [x])
```

go back up

```haskell
data Free f a where
  Pure :: a -> Free f a
  Ap :: Free f (a -> b) -> f a -> Free f b

gfoldl f z = foldAp f z . gtraverse (liftAp . I)

foldAp
  :: (forall d b. Data d => c (d -> b) -> d -> c b)
  -> (forall g. g -> c g)
  -> Ap I a -> c a
foldAp f z (Pure x) = z x
foldAp f z (Ap (I x) k) = (foldAp f z k) `f` x
```

Many Data instances

```haskell
class Data a where
  gtraverse
    :: Applicative c
    => (forall d . Data d => d -> c d)
    -> a -> c a

instance (Data a, Data b) => Data (a,b) where
  gtraverse f (a,b) = (,) <$> f a <*> f b

instance Data a => Data (a,b) where
  gtraverse f (a,b) = (,) <$> f a <*> pure b

instance Data b => Data (a,b) where
  gtraverse f (a,b) = (,) <$> pure a <*> f b
```

Many Data instances

> «All problems in Haskell can be solved by adding another type parameter»

```
class Data (c :: * -> Constraint) a where
  gtraverse
    :: (Applicative f)
    => (forall d . c d => d -> f d)
    -> a -> f a
```