Fly Like an Arrow

Jonathan Fischoff ✉ jonathangfischoff@gmail.com

March 26, 2012

The great thing about Arrows is you can write code that works for morphisms in different categories. For example, you can write code for functions and later use monad actions, or Kleisli arrows, instead. This is useful for error handling, and of course, adding `IO`.

If the underlying category uses isomorphisms (things with inverses) exclusively then it is called a *groupoid*. Groupoids cause cracks to show in the `Arrow` abstraction. `Arrow` assumes that you can lift any function into the category you are writing code for, by requiring a definition for `arr :: (b -> c) -> a b c` function. This is out for groupoids, because not all functions are isomorphisms.

To remedy this, among other issues, Adam Megacz came up with. Generalized Arrows..

In *Dagger Traced Symmetric Monoidal Categories and Reversible Programming* the authors show how to construct an reversible language out of the sum and product types along with related combinators to form a commutative semiring, at the type level. Both approaches are similar.

Error handling and *partial isomorphisms* are possible with Generalized Arrows. However, I find the algebraic approach of *DTSMCRP* more elegant. So I am going to try to get the same combinators as *DTSMCRP* but for an arbitrary category, as I would with Generalized Arrows.

This is a Literate Haskell file, which means it can be executed as Haskell code. First, I need to start with a simple Haskell preamble.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
 -- Categorical semirings (my term, but maybe the correct one) are an alternative to Arrows, but play n
 -- See the source or the latex source for more background.
module Data.Semiring (
    -- * Endofunctors for construction
    Ctor (..)
    -- ** first/right like functions
  , promote
```

```
  , swap_promote
    -- * Laws (Axioms) for building algebraic structures
  , Absorbs (..)
  , Assocative (..)
  , Commutative (..)
  , Annihilates (..)
  , Distributes (..)
    -- * Categorical Algebraic Structures
  , Monoidial
  , CommutativeMonoidial
  , Semiring
    -- * Arrow like functions for semiring categories
  , first
  , second
  , left
  , right
    -- * A groupoid class that is a category. Maybe this is a bad idea?
  , Groupoid (..)
  , Iso (..)
    -- * Alegraic laws as isomorphism for groupoid instances
  , biject_sum_absorb
  , biject_sum_assoc
  , biject_product_absorb
  , biject_product_assoc
  , biject_distributes
  , kbiject_sum_absorb
  , kbiject_sum_assoc
  , kbiject_product_absorb
  , kbiject_product_assoc
  , kbiject_distributes
  ) where
import Prelude hiding ((∘), id)
import Control.Category ((∘), id, Category (..))
import Data.Void (Void)
import Control.Arrow (Kleisli (..))
import Generics.Pointless.MonadCombinators (mfuse)
import Control.Monad (liftM)
import Control.Newtype
```

I start with an abstraction for both sum and product constructors.

```
-- An endofunctor for combining two morphisms
class Category k => Ctor k constr | constr -> k where
  selfmap :: k a b -> k c d -> k (constr a c) (constr b d)
```

With Ctor I can write a generic `first` or `left`

```
-- construct a new morphism with identity
promote :: Ctor k op => k a b -> k (op a c) (op b c)
promote = flip selfmap id

-- construct a new morphism with identity with the arguments reversed
swap_promote :: Ctor k op => k a b -> k (op c a) (op c b)
swap_promote = selfmap id
```

It is probably not clear at this point but depending on the type of *op* we can get either the `Arrow` $***$ or the `ArrowChoice` $|||$ function. If we make a semiring we can get them both. That's what we are going to do.

I use type classes to encode the algebraic laws of semirings, with a class per law.

```
-- The absorbtion law => x + 0 <-> x
class Ctor k op => Absorbs k op id | op -> k, op -> id where
  absorb :: k (op id a) a
  unabsorb :: k a (op id a)


-- The commutative law => x + y <-> y + x
class Ctor k op => Commutative k op | op -> k where
  commute :: k (op a b) (op b a)


-- The assocative law => (x + y) + z <-> x + (y + z)
class Ctor k op => Assocative k op | op -> k where
  assoc   :: k (op (op a b) c) (op a (op b c))
  unassoc :: k (op a (op b c)) (op (op a b) c)


-- The annihilation law => 0 * x <-> 0
class Ctor k op => Annihilates k op zero | op zero -> k where
  annihilates :: k (op zero a) zero


-- The distribution law => (a + b) * c <-> (a * c) + (b * c)
class (Ctor k add, Ctor k multi) => Distributes k add multi | add multi -> k where
  distribute :: k (multi (add a b) c) (add (multi a c) (multi b c))
  undistribute :: k (add (multi a c) (multi b c)) (multi (add a b) c)
```

I collect these into groups of laws to make different algebraic structures.

```
-- Monoidial Category class
class (Assocative k dot, Absorbs k dot id) ⇒
  Monoidial k dot id | dot id → k where


-- Commutative Monoidial Category class
class (Monoidial k dot id, Commutative k dot) ⇒
  CommutativeMonoidial k dot id | dot id → k where


-- Semiring Category class
class (CommutativeMonoidial k add zero,
  CommutativeMonoidial k multi one,
  Annihilates k multi zero,
  Distributes k add multi) ⇒
  Semiring k add zero multi one | add zero multi one → k where
```

From which I regain `Arrow` and `ArrowChoice` functionality. Although, because of `promote`, I already had this capability.

```
  -- Apply the multi monoid operator to the morphism and identity
first :: Semiring a add zero multi one ⇒ a b c → a (multi b d) (multi c d)
first = promote
  -- Apply the multi monoid operator to identity and the morphism
second :: Semiring a add zero multi one ⇒ a b c → a (multi d b) (multi d c)
second = swap_promote
  -- Apply the add monoid operator to the morphism and identity
left :: Semiring a add zero multi one ⇒ a b c → a (add b d) (add c d)
left = promote
  -- Apply the add monoid operator to identity and the morphism
right :: Semiring a add zero multi one ⇒ a b c → a (add d b) (add d c)
right = swap_promote
```

Many of the Generic Arrow functions can be included through absorption (`cancel`, `uncancel`) and commutativity (`swap`). I'm not interested in adding looping at this point.

This also makes clear the relationship between `Arrow` and `ArrowChoice` as has been noted else where. Basically the same thing with a different endofunctor or constructor (`Arrow` uses product types, `ArrowChoice` uses sum types) as the monoid operator of a type level commutative monoid.

Making instances is a little onerous because of the use of multiparameter classes and the functional dependencies I have chosen. When I begin actually using these classes, it could result in massive changes. I am open to any suggestions on better designs.

The rest of the code is basically boilerplate. I used Djinn to write some of the functions (hopefully they work :)).

1 Small Category Instances

1.1 Function Instances

1.1.1 Sum Commutative Monoid Instances

```
  -- Sugar
type + = Either
type 0 = Void
  -- Instances
instance Ctor (→) + where
  selfmap f g = either (Left ∘ f) (Right ∘ g)
instance Absorbs (→) + 0 where
  absorb (Right x) = x
  absorb _ = error "Absorbs,->,Sum,Zero Semiring.lhs absorb:impossible!"
  unabsorb = Right
instance Assocative (→) + where
  assoc   = either (either Left (Right ∘ Left)) (Right ∘ Right)
  unassoc = either (Left ∘ Left) (either (Left ∘ Right) Right)
instance Monoidial (→) + 0 where
instance Commutative (→) + where
  commute = either Right Left
instance CommutativeMonoidial (→) + 0 where
```

1.1.2 Product Commutative Monoid Instances

```
type ∗ = (,)
type 1 = ()
  -- Instances
instance Ctor (→) ∗ where
  selfmap f g (x, y) = (f x, g y)
instance Absorbs (→) ∗ 1 where
  absorb ((), x) = x
  unabsorb x = ((), x)
instance Assocative (→) ∗ where
  assoc   ((x, y), z) = (x, (y, z))
  unassoc (x, (y, z)) = ((x, y), z)
instance Monoidial (→) ∗ 1 where
instance Commutative (→) ∗ where
```

$commute\ (x, y) = (y, x)$
instance $CommutativeMonoidial\ (\rightarrow)\ *\ 1$ **where**

1.1.3 Function Semiring Instance

instance $Annihilates\ (\rightarrow)\ *\ 0$ **where**
  $annihilates\ (\_, \_) = \bot$
instance $Distributes\ (\rightarrow)\ +\ *$ **where**
  $distribute\ (Left\ x, z) = Left\ (x, z)$
  $distribute\ (Right\ y, z) = Right\ (y, z)$
  $undistribute\ (Left\ (x, z)) = (Left\ x, z)$
  $undistribute\ (Right\ (y, z)) = (Right\ y, z)$
instance $Semiring\ (\rightarrow)\ +\ 0\ *\ 1$ **where**

1.2 Kleisli Instances

The functional dependencies of the classes require alternate versions of the sum and product types used for $\rightarrow$ *instances*.

1.3 Sum Commutative Monoid Instances

data $+\ a\ b = KLeft\ a \mid KRight\ b$
newtype $0 = KZ\ Void$
  -- Instances
instance $Monad\ m \Rightarrow Ctor\ (Kleisli\ m)\ +$ **where**
  $selfmap\ (Kleisli\ f)\ (Kleisli\ g) = Kleisli\ \$$
    $\lambda e \rightarrow$ **case** e **of**
      $KLeft\ x \rightarrow KLeft\ `liftM`\ f\ x$
      $KRight\ x \rightarrow KRight\ `liftM`\ g\ x$
instance $Monad\ m \Rightarrow Absorbs\ (Kleisli\ m)\ +\ 0$ **where**
  $absorb = Kleisli\ \$\ \lambda e \rightarrow$ **case** e **of**
    $KRight\ x \rightarrow return\ x$
    $\_ \rightarrow error$ `"Absorbs,Kleisli,KSum,KZero Semiring.lhs absorb:impossible!"`
  $unabsorb = Kleisli\ \$\ \lambda x \rightarrow return\ \$\ KRight\ x$
instance $Monad\ m \Rightarrow Assocative\ (Kleisli\ m)\ +$ **where**
  $assoc = Kleisli\ \$\ \lambda e \rightarrow$ **case** e **of**
      $KLeft\ x \rightarrow$ **case** x **of**
          $KLeft\ y \rightarrow return\ \$\ KLeft\ y$
          $KRight\ y \rightarrow return\ \$\ KRight\ \$\ KLeft\ y$
      $KRight\ x \rightarrow return\ \$\ KRight\ \$\ KRight\ x$
  $unassoc = Kleisli\ \$\ \lambda e \rightarrow$ **case** e **of**
      $KLeft\ x \rightarrow return\ \$\ KLeft\ \$\ KLeft\ x$

```
            KRight x → case x of
                         KLeft y   → return $ KLeft $ KRight y
                         KRight y → return $ KRight y
instance Monad m ⇒ Monoidial (Kleisli m) + 0 where
instance Monad m ⇒ Commutative (Kleisli m) + where
  commute = Kleisli $ λe → case e of
    KLeft x → return $ KRight x
    KRight x → return $ KLeft x
instance Monad m ⇒ CommutativeMonoidial (Kleisli m) + 0 where
```

1.3.1 Product Commutative Monoid Instances

```
data ∗ a b = KP a b
newtype 1 = KO ()
  -- Instances
instance Monad m ⇒ Ctor (Kleisli m) ∗ where
  selfmap (Kleisli f) (Kleisli g) = Kleisli $
    λ(KP x y) → uncurry KP ‘liftM‘ mfuse (f x, g y)
instance Monad m ⇒ Absorbs (Kleisli m) ∗ 1 where
  absorb = Kleisli $ λ(KP (KO ()) x) → return x
  unabsorb = Kleisli $ λx → return $ KP (KO ()) x
instance Monad m ⇒ Assocative (Kleisli m) ∗ where
  assoc = Kleisli $ λ(KP (KP x y) z) → return $ KP x (KP y z)
  unassoc = Kleisli $ λ(KP x (KP y z)) → return $ KP (KP x y) z
instance Monad m ⇒ Monoidial (Kleisli m) ∗ 1 where
instance Monad m ⇒ Commutative (Kleisli m) ∗ where
  commute = Kleisli $ λ(KP x y) → return $ KP y x
instance Monad m ⇒ CommutativeMonoidial (Kleisli m) ∗ 1 where
```

1.3.2 Function Semiring Instance

```
instance Monad m ⇒ Annihilates (Kleisli m) ∗ 0 where
  annihilates = Kleisli $ λ(KP _ _) → return ⊥
instance Monad m ⇒ Distributes (Kleisli m) + ∗ where
  distribute = Kleisli $ λe → case e of
          KP (KLeft x) z → return $ KLeft $ KP x z
          KP (KRight y) z → return $ KRight $ KP y z
  undistribute = Kleisli $ λe → case e of
    KLeft (KP x z) → return $ KP (KLeft x)   z
    KRight (KP x z) → return $ KP (KRight x) z
instance Monad m ⇒ Semiring (Kleisli m) + 0 ∗ 1 where
```

2 Groupoid Class

class $(Category\ g) \Rightarrow Groupoid\ g$ **where**
  $inv :: g\ a\ b \rightarrow g\ b\ a$

3 Groupoid Instances

data $Iso\ k\ a\ b = Iso\ \{$
    $embed :: k\ a\ b,$
    $project :: k\ b\ a$
  $\}$

instance $(Category\ k) \Rightarrow Category\ (Iso\ k)$ **where**
  $id = Iso\ id\ id$
  $(Iso\ f\ g) \circ (Iso\ h\ i) = Iso\ (f \circ h)\ (i \circ g)$

instance $Newtype\ (Iso\ k\ a\ b)\ (k\ a\ b, k\ b\ a)$ **where**
  $pack\ (f, g) \qquad = Iso\ f\ g$
  $unpack\ (Iso\ f\ g) = (f, g)$

instance $(Category\ k) \Rightarrow Groupoid\ (Iso\ k)$ **where**
  $inv\ (Iso\ f\ g) = Iso\ g\ f$

3.1 Helper Code

type $Biject = Iso\ (\rightarrow)$
type $KBiject\ m = Iso\ (Kleisli\ m)$

$(< - >) :: k\ a\ b \rightarrow k\ b\ a \rightarrow Iso\ k\ a\ b$
$(< - >) = Iso$

3.2 Groupoid Semirings Instances

3.3 Groupoid with a Function as the base category

3.3.1 Sum Commutative Monoid Instances

data $+\ a\ b = BLeft\ a \mid BRight\ b$
newtype $0 = BZ\ Void$

instance $Ctor\ Biject\ +$ **where**
  $selfmap\ f\ g = fw < - > bk$ **where**
    $fw\ (BLeft\ x) = BLeft\ \$\ embed\ f\ x$
    $fw\ (BRight\ x) = BRight\ \$\ embed\ g\ x$
    $bk\ (BLeft\ x) = BLeft\ \$\ project\ f\ x$

```
    bk (BRight x) = BRight $ project g x
instance Absorbs Biject + 0 where
  absorb = biject_sum_absorb
  unabsorb = inv biject_sum_absorb

biject_sum_absorb :: Biject (+ 0 a) a
biject_sum_absorb = fw <-> bk where
  fw (BRight x) = x
  fw _ = error "biject_sum_absorb fw: impossible"
  bk = BRight

instance Assocative Biject + where
  assoc = biject_sum_assoc
  unassoc = inv biject_sum_assoc

biject_sum_assoc :: Biject (+ (+ a b) c) (+ a (+ b c))
biject_sum_assoc = fw <-> bk where
  fw (BLeft (BLeft x)) = BLeft x
  fw (BLeft (BRight x)) = BRight $ BLeft x
  fw (BRight x)         = BRight $ BRight x

  bk (BLeft x)               = BLeft (BLeft x)
  bk (BRight (BLeft x)) = BLeft (BRight x)
  bk (BRight (BRight x)) = BRight x

instance Monoidial Biject + 0 where

instance Commutative Biject + where
  commute = fw <-> bk where
    fw (BRight x) = BLeft x
    fw (BLeft x) = BRight x

    bk (BRight x) = BLeft x
    bk (BLeft x) = BRight x

instance CommutativeMonoidial Biject + 0 where
```

3.3.2 Product Commutative Monoid Instances

```
data * a b = BP a b
newtype 1 = BO ()
  -- Instances
instance Ctor Biject * where
  selfmap (Iso f_fw f_bk) (Iso g_fw g_bk) =
    Iso (λ(BP x y) → BP (f_fw x) (g_fw y)) (λ(BP x y) → BP (f_bk x) (g_bk y))
instance Absorbs Biject * 1 where
  absorb = biject_product_absorb
  unabsorb = inv biject_product_absorb

biject_product_absorb :: Biject (* 1 a) a
biject_product_absorb = fw <-> bk where
```

$fw\ (BP\ (BO\ ())\ x) = x$
$bk\ x = BP\ (BO\ ())\ x$
instance *Assocative Biject* $*$ **where**
$assoc = biject\_product\_assoc$
$unassoc = inv\ biject\_product\_assoc$

$biject\_product\_assoc :: Biject\ (*\ (*\ a\ b)\ c)\ (*\ a\ (*\ b\ c))$
$biject\_product\_assoc = fw <-> bk$ **where**
$fw\ (BP\ (BP\ x\ y)\ z) = BP\ x\ (BP\ y\ z)$
$bk\ (BP\ x\ (BP\ y\ z)) = BP\ (BP\ x\ y)\ z$
instance *Monoidial Biject* $*$ 1 **where**
instance *Commutative Biject* $*$ **where**
$commute = (\lambda(BP\ x\ y) \rightarrow BP\ y\ x) <-> (\lambda(BP\ x\ y) \rightarrow BP\ y\ x)$
instance *CommutativeMonoidial Biject* $*$ 1 **where**

3.3.3 Semiring Instance

instance *Annihilates Biject* $*$ 0 **where**
$annihilates = (\lambda(BP\ \_\ \_) \rightarrow \bot) <-> (\text{`}BP\text{`}\bot)$
instance *Distributes Biject* $+$ $*$ **where**
$distribute = biject\_distributes$
$undistribute = inv\ biject\_distributes$

$biject\_distributes :: Biject\ (*\ (+\ a\ b)\ c)\ (+\ (*\ a\ c)\ (*\ b\ c))$
$biject\_distributes = fw <-> bk$ **where**
$fw\ (BP\ (BLeft\ x)\ z) = BLeft\ (BP\ x\ z)$
$fw\ (BP\ (BRight\ y)\ z) = BRight\ (BP\ y\ z)$
$bk\ (BLeft\ (BP\ x\ z)) = BP\ (BLeft\ x)\ z$
$bk\ (BRight\ (BP\ y\ z)) = BP\ (BRight\ y)\ z$
instance *Semiring Biject* $+$ 0 $*$ 1 **where**

3.4 Groupoid with a Klesli arrow as the base category

3.4.1 Sum Commutative Monoid Instances

data $+\ a\ b = KBLeft\ a \mid KBRight\ b$
newtype $0 = KBZ\ Void$
-- Instances
instance $Monad\ m \Rightarrow Ctor\ (KBiject\ m)\ +$ **where**
$selfmap\ f\ g = fw <-> bk$ **where**
$fw = Kleisli\ \$\ run\_pair\ (embed\ f)\ (embed\ g)$
$bk = Kleisli\ \$\ run\_pair\ (project\ f)\ (project\ g)$
$run\_pair\ t\ \_\ (KBLeft\ x) = KBLeft\ \text{`}liftM\text{`}\ runKleisli\ t\ x$

```
    run_pair _ u (KBRight x) = KBRight `liftM` runKleisli u x
instance Monad m => Absorbs (KBiject m) + 0 where
  absorb = kbiject_sum_absorb
  unabsorb = inv kbiject_sum_absorb
kbiject_sum_absorb :: Monad m => (KBiject m) (+ 0 a) a
kbiject_sum_absorb = fw <-> bk where
  fw = Kleisli $ \e -> case e of
    KBRight x -> return x
    _ -> error "kbiject_sum_absorb fw: impossible"
  bk = Kleisli $ \x -> return $ KBRight x
instance Monad m => Assocative (KBiject m) + where
  assoc = kbiject_sum_assoc
  unassoc = inv kbiject_sum_assoc
kbiject_sum_assoc :: Monad m => (KBiject m) (+ (+ a b) c) (+ a (+ b c))
kbiject_sum_assoc = fw <-> bk where
  fw = Kleisli $ \e -> case e of
    KBLeft x -> case x of
                  KBLeft y -> return $ KBLeft y
                  KBRight y -> return $ KBRight $ KBLeft y
    KBRight x -> return $ KBRight $ KBRight x
  bk = Kleisli $ \e -> case e of
    KBLeft x -> return $ KBLeft $ KBLeft x
    KBRight x -> case x of
                  KBLeft y -> return $ KBLeft $ KBRight y
                  KBRight y -> return $ KBRight y
instance Monad m => Monoidial (KBiject m) + 0 where
instance Monad m => Commutative (KBiject m) + where
  commute = fw <-> fw where
    fw = Kleisli $ \e -> case e of
           KBLeft x -> return $ KBRight x
           KBRight x -> return $ KBLeft x
instance (Monad m) => CommutativeMonoidial (KBiject m) + 0 where
```

3.4.2 Product Commutative Monoid Instances

```
data * a b = KBP a b
newtype 1 = KBO ()
  -- Instances
instance Monad m => Ctor (KBiject m) * where
  selfmap f g = fw <-> bk where
    fw = Kleisli $ run_pair (embed f) (embed g)
    bk = Kleisli $ run_pair (project f) (project g)
    run_pair t u (KBP x y) =
```

```
    uncurry KBP `liftM` mfuse (runKleisli t x, runKleisli u y)
instance Monad m ⇒ Absorbs (KBiject m) ∗ 1 where
  absorb = kbiject_product_absorb
  unabsorb = inv kbiject_product_absorb

kbiject_product_absorb :: Monad m ⇒ (KBiject m) (∗ 1 a) a
kbiject_product_absorb = fw < − > bk where
  fw = Kleisli $ λ(KBP (KBO ()) x) → return x
  bk = Kleisli $ λx → return $ KBP (KBO ()) x

instance Monad m ⇒ Assocative (KBiject m) ∗ where
  assoc = kbiject_product_assoc
  unassoc = inv kbiject_product_assoc

kbiject_product_assoc :: Monad m
   ⇒ (KBiject m) (∗ (∗ a b) c) (∗ a (∗ b c))
kbiject_product_assoc = fw < − > bk where
  fw = Kleisli $ λ(KBP (KBP f g) h) → return $ KBP f (KBP g h)
  bk = Kleisli $ λ(KBP f (KBP g h)) → return $ KBP (KBP f g) h

instance Monad m ⇒ Monoidial (KBiject m) ∗ 1 where

instance Monad m ⇒ Commutative (KBiject m) ∗ where
  commute = fw < − > fw where
    fw = Kleisli $ λ(KBP x y) → return $ KBP y x

instance (Monad m) ⇒ CommutativeMonoidial (KBiject m) ∗ 1 where
```

3.4.3 Semiring Instance

```
instance (Monad m) ⇒ Annihilates (KBiject m) ∗ 0 where
  annihilates = fw < − > bk where
    fw = Kleisli $ λ(KBP _ _) → return ⊥
    bk = Kleisli $ λx → return $ KBP x ⊥

instance (Monad m) ⇒ Distributes (KBiject m) + ∗ where
  distribute = kbiject_distributes
  undistribute = inv kbiject_distributes

kbiject_distributes :: Monad m
     ⇒ (KBiject m) (∗ (+ a b) c) (+ (∗ a c) (∗ b c))
kbiject_distributes = fw < − > bk where
  fw = Kleisli $ λe → case e of
    KBP (KBLeft x) z → return $ KBLeft (KBP x z)
    KBP (KBRight y) z → return $ KBRight (KBP y z)

  bk = Kleisli $ λe → case e of
    KBLeft (KBP x z) → return $ KBP (KBLeft x) z
    KBRight (KBP y z) → return $ KBP (KBRight y) z

instance Monad m ⇒ Semiring (KBiject m) + 0 ∗ 1 where
```