Building a Better Arrow

Jonathan Fischoff

March 25, 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 only uses isomorphisms (things with inverses) 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 , FunctionalDependencies , TypeOperators , TypeSynonym
module Data.Semiring 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, liftM2)
import qualified Data.Groupoid as DataGroupoid (Groupoid (..))
import qualified Data.Groupoid.Isomorphism as DataGroupoid (Iso (..))
import Data.Functor.Bind (Bind (..))
import Control.Newtype
```

I start with an umbrella class for both sum and product constructors.

```haskell
class Category k => Ctor k constr | constr -> k where
  selfmap :: k a b -> k c d -> k (constr a c) (constr b d)
```

An `Ctor` is really a binary endofunctor, because it takes a morphism from a category to the same category.
I now write the following pretty general functions.

```haskell
promote :: Ctor k op => k a b -> k (op a c) (op b c)
promote = (flip selfmap) id

swapPromote :: Ctor k op => k a b -> k (op c a) (op c b)
swapPromote = selfmap id
```

It is probably not clear at this point but depending on the type of /sl 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.

Now I can make the type classes to encode the algebraic laws of semirings. I make a class for each law.

$$a + 0 \leftrightarrow a$$

```haskell
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)
```

$$a + b \leftrightarrow b + a$$

```haskell
class Ctor k op => Communative k op | op -> k where
  commute :: k (op a b) (op b a)
```

$$(a + b) + c \leftrightarrow a + (b + c)$$

```haskell
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)
```

$$0 * a \leftrightarrow 0$$

```haskell
class Ctor k op => Annihilates k op zero | op zero -> k where
  annihilates :: k (op zero a) zero
```

$$(a + b) * c \leftrightarrow (a * c) + (b * c)$$

```haskell
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)
```

Now I can collect these into groups of laws for different algebraic structures I care about.

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Assocative}\ k\ \mathit{dot}, \mathit{Absorbs}\ k\ \mathit{dot}\ \mathit{id}) \Rightarrow \\
&\quad \mathit{Monoidial}\ k\ \mathit{dot}\ \mathit{id} \mid \mathit{dot}\ \mathit{id} \rightarrow k\ \textbf{where}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Monoidial}\ k\ \mathit{dot}\ \mathit{id}, \mathit{Communative}\ k\ \mathit{dot}) \Rightarrow \\
&\quad \mathit{CommunativeMonoidial}\ k\ \mathit{dot}\ \mathit{id} \mid \mathit{dot}\ \mathit{id} \rightarrow k\ \textbf{where}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ (\mathit{CommunativeMonoidial}\ k\ \mathit{add}\ \mathit{zero}, \\
&\quad \mathit{CommunativeMonoidial}\ k\ \mathit{multi}\ \mathit{one}, \\
&\quad \mathit{Annihilates}\ k\ \mathit{multi}\ \mathit{zero}, \\
&\quad \mathit{Distributes}\ k\ \mathit{add}\ \mathit{multi}) \Rightarrow \\
&\quad \mathit{Semiring}\ k\ \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \mid \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \rightarrow k\ \textbf{where}
\end{aligned}
$$

From which I regain `Arrow` functionality.

$$
\begin{aligned}
&\mathit{first} :: \mathit{Semiring}\ a\ \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \Rightarrow a\ b\ c \rightarrow a\ (\mathit{multi}\ b\ d)\ (\mathit{multi}\ c\ d) \\
&\mathit{first} = \mathit{promote}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{second} :: \mathit{Semiring}\ a\ \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \Rightarrow a\ b\ c \rightarrow a\ (\mathit{multi}\ d\ b)\ (\mathit{multi}\ d\ c) \\
&\mathit{second} = \mathit{swapPromote}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{left} :: \mathit{Semiring}\ a\ \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \Rightarrow a\ b\ c \rightarrow a\ (\mathit{add}\ b\ d)\ (\mathit{add}\ c\ d) \\
&\mathit{left} = \mathit{promote}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{right} :: \mathit{Semiring}\ a\ \mathit{add}\ \mathit{zero}\ \mathit{multi}\ \mathit{one} \Rightarrow a\ b\ c \rightarrow a\ (\mathit{add}\ d\ b)\ (\mathit{add}\ d\ c) \\
&\mathit{right} = \mathit{swapPromote}
\end{aligned}
$$

Many of the Generic Arrow functions can be included through absorption (`cancel`, `uncancel`) and commutativity (`swap`).

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

Two important and instances are sum and product, or in Haskell parlance `(,)` tuples and `Either` respectively.

1 Small Category Instances

1.1 Function Instances

1.1.1 Sum Communative Monoid Instances

-- Sugar
type $\sum = \mathit{Either}$
type $0 = \mathit{Void}$
-- Instances
instance $\mathit{Ctor}\ (\rightarrow)\ \sum$ **where**
$\quad \mathit{selfmap}\ f\ g = \mathit{either}\ (\mathit{Left} \circ f)\ (\mathit{Right} \circ g)$
instance $\mathit{Absorbs}\ (\rightarrow)\ \sum\ 0$ **where**
$\quad \mathit{absorb}\ (\mathit{Right}\ x) = x$
$\quad \mathit{unabsorb}\ x = \mathit{Right}\ x$
instance $\mathit{Assocative}\ (\rightarrow)\ \sum$ **where**
$\quad \mathit{assoc} \quad = \mathit{either}\ (\mathit{either}\ (\mathit{Left})\ (\mathit{Right} \circ \mathit{Left}))\ (\mathit{Right} \circ \mathit{Right})$
$\quad \mathit{unassoc} = \mathit{either}\ (\mathit{Left} \circ \mathit{Left})\ (\mathit{either}\ (\mathit{Left} \circ \mathit{Right})\ (\mathit{Right}))$
instance $\mathit{Monoidial}\ (\rightarrow)\ \sum\ 0$ **where**
instance $\mathit{Communative}\ (\rightarrow)\ \sum$ **where**
$\quad \mathit{commute} = \mathit{either}\ (\mathit{Right})\ (\mathit{Left})$
instance $\mathit{CommunativeMonoidial}\ (\rightarrow)\ \sum\ 0$ **where**

1.1.2 Product Commutative Monoid Instances

type $\prod = (,)$
type $1 = ()$
-- Instances
instance $\mathit{Ctor}\ (\rightarrow)\ \prod$ **where**
$\quad \mathit{selfmap}\ f\ g\ (x, y) = (f\ x, g\ y)$
instance $\mathit{Absorbs}\ (\rightarrow)\ \prod\ 1$ **where**
$\quad \mathit{absorb}\ ((), x) = x$
$\quad \mathit{unabsorb}\ x = ((), x)$
instance $\mathit{Assocative}\ (\rightarrow)\ \prod$ **where**
$\quad \mathit{assoc} \quad ((x, y), z) = (x, (y, z))$
$\quad \mathit{unassoc}\ (x, (y, z)) = ((x, y), z)$
instance $\mathit{Monoidial}\ (\rightarrow)\ \prod\ 1$ **where**
instance $\mathit{Communative}\ (\rightarrow)\ \prod$ **where**
$\quad \mathit{commute}\ (x, y) = (y, x)$

instance *CommunativeMonoidial* $(\to)\ \prod\ 1$ **where**

1.1.3 Function Semiring Instance

instance *Annihilates* $(\to)\ \prod\ 0$ **where**
  annihilates $(\bot, x) = \bot$
instance *Distributes* $(\to)\ \sum\ \prod$ **where**
  distribute $(\mathit{Left}\ x, z) = \mathit{Left}\ (x, z)$
  distribute $(\mathit{Right}\ y, z) = \mathit{Right}\ (y, z)$
  undistribute $(\mathit{Left}\ (x, z)) = (\mathit{Left}\ x, z)$
  undistribute $(\mathit{Right}\ (y, z)) = (\mathit{Right}\ y, z)$
instance *Semiring* $(\to)\ \sum\ 0\ \prod\ 1$ **where**

1.2 Kleisli Instances

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

1.3 Sum Communative Monoid Instances

```
data ∑ a b = KLeft a | KRight b
newtype 0 = KZ Void
  -- Instances
instance Monad m ⇒ Ctor (Kleisli m) ∑ where
  selfmap (Kleisli f) (Kleisli g) = Kleisli $
    λe → case e of
        KLeft x → KLeft ‘liftM‘ f x
        KRight x → KRight ‘liftM‘ g x
instance Monad m ⇒ Absorbs (Kleisli m) ∑ 0 where
  absorb = Kleisli $ λ(KRight x) → return x
  unabsorb = Kleisli $ λx → return $ KRight x
instance Monad m ⇒ Assocative (Kleisli m) ∑ where
  assoc = Kleisli $ λe → case e of
            KLeft x → case x of
                        KLeft y → return $ KLeft y
                        KRight y → return $ KRight $ KLeft y
            KRight x → return $ KRight $ KRight x
  unassoc = Kleisli $ λe → case e of
            KLeft x → 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 ⇒ Communative (Kleisli m) ∑ where
  commute = Kleisli $ λx → case x of
    KLeft x → return $ KRight x
    KRight x → return $ KLeft x
instance Monad m ⇒ CommunativeMonoidial (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 ⇒ Communative (Kleisli m) ∏ where
  commute = Kleisli $ λ(KP x y) → return $ KP y x
instance Monad m ⇒ CommunativeMonoidial (Kleisli m) ∏ 1 where
```

1.3.2 Function Semiring Instance

```
instance Monad m ⇒ Annihilates (Kleisli m) ∏ 0 where
  annihilates = Kleisli $ λ(KP ⊥ x) → 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 k, Category (t k)) ⇒ Groupoid t k | k → t where
  inv :: t k a b → t k b a
```

3 Groupoid Instances

```
data Iso k a b = Iso {
    embed :: k a b,
    project :: k b a
  }
instance (Category k) ⇒ Category (Iso k) where
  id = Iso id id
  (Iso f g) ∘ (Iso h i) = Iso (f ∘ h) (i ∘ g)
instance Newtype (Iso k a b) (k a b, k b a) where
  pack (f, g)         = Iso f g
  unpack (Iso f g) = (f, g)
instance (Category k) ⇒ Groupoid Iso k where
  inv (Iso f g) = Iso g f
```

3.1 Helper Code

```
type Biject = Iso (→)
type KBiject m = Iso (Kleisli m)
(< − >) = Iso
```

3.2 Groupoid Semirings Instances

3.3 Groupoid with a Function as the base category

3.3.1 Sum Communative Monoid Instances

```
data BSum a b = BLeft a | BRight b
newtype BZero = BZ Void
instance Ctor Biject BSum 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 BSum BZero where
  absorb = biject_sum_absorb
  unabsorb = inv biject_sum_absorb
biject_sum_absorb :: Biject (BSum BZero a) (a)
biject_sum_absorb = fw <-> bk where
  fw (BRight x) = x
  bk x = BRight x
instance Assocative Biject BSum where
  assoc = biject_sum_assoc
  unassoc = inv biject_sum_assoc
biject_sum_assoc :: Biject (BSum (BSum a b) c) (BSum a (BSum 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 BSum BZero where
instance Communative Biject BSum 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 CommunativeMonoidial Biject BSum BZero where
```

3.3.2 Product Communative Monoid Instances

```
data BProduct a b = BP a b
newtype BOne = BO ()
  -- Instances
instance Ctor Biject BProduct 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 BProduct BOne where
  absorb = biject_product_absorb_iso
  unabsorb = inv biject_product_absorb_iso
biject_product_absorb_iso :: Biject (BProduct BOne a) a
biject_product_absorb_iso = fw <-> bk where
  fw (BP (BO ()) x) = x
```

```
  bk x = (BP (BO ()) x)
instance Assocative Biject BProduct where
  assoc = biject_product_assocate_iso
  unassoc = inv biject_product_assocate_iso
biject_product_assocate_iso :: Biject (BProduct (BProduct a b) c) (BProduct a (BProduct b c))
biject_product_assocate_iso = 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 BProduct BOne where
instance Communative Biject BProduct where
  commute = (λ(BP x y) → BP y x) <-> (λ(BP x y) → BP y x)
instance CommunativeMonoidial Biject BProduct BOne where
```

3.3.3 Semiring Instance

```
instance Annihilates Biject BProduct BZero where
  annihilates = (λ(BP ⊥ x) → ⊥) <-> (λx → BP x ⊥)
instance Distributes Biject BSum BProduct where
  distribute = biject_distributes_iso
  undistribute = inv biject_distributes_iso
biject_distributes_iso :: Biject (BProduct (BSum a b) c) (BSum (BProduct a c) (BProduct b c))
biject_distributes_iso = 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 BSum BZero BProduct BOne where
```

3.4 Groupoid with a Klesli arrow as the base category

3.4.1 Sum Communative Monoid Instances

```
data KBSum a b = KBLeft a | KBRight b
newtype KBZero = KBZ Void
  -- Instances
instance (Monad m) ⇒ Ctor (KBiject m) KBSum where
  selfmap f g = fw <-> bk where
    fw = Kleisli $
      λe → case e of
             KBLeft x → KBLeft `liftM` (runKleisli $ embed f) x
             KBRight x → KBRight `liftM` (runKleisli $ embed g) x
```

```
    bk = Kleisli $
      λe → case e of
              KBLeft x → KBLeft `liftM` (runKleisli $ project f) x
              KBRight x → KBRight `liftM` (runKleisli $ project g) x
instance (Monad m) ⇒ Absorbs (KBiject m) KBSum KBZero where
  absorb = kbiject_sum_absorb
  unabsorb = inv kbiject_sum_absorb
kbiject_sum_absorb :: (Monad m) ⇒ (KBiject m) (KBSum KBZero a) (a)
kbiject_sum_absorb = fw < − > bk where
  fw = Kleisli $ λ(KBRight x) → return x
  bk = Kleisli $ λx → return $ KBRight x
instance (Monad m) ⇒ Assocative (KBiject m) KBSum where
  assoc = kbiject_sum_assoc
  unassoc = inv kbiject_sum_assoc
kbiject_sum_assoc :: (Monad m) ⇒ (KBiject m) (KBSum (KBSum a b) c) (KBSum a (KBSum b c))
kbiject_sum_assoc = fw < − > bk where
  fw = Kleisli $ λe → 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 $ λe → 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) KBSum KBZero where
instance (Monad m) ⇒ Communative (KBiject m) KBSum where
  commute = fw < − > fw where
    fw = Kleisli $ λx → case x of
              KBLeft x → return $ KBRight x
              KBRight x → return $ KBLeft x
instance (Monad m) ⇒ CommunativeMonoidial (KBiject m) KBSum KBZero where
```

3.4.2 Product Communative Monoid Instances

```
data KBProduct a b = KBP a b
newtype KBOne = KBO ()
  -- Instances
instance (Monad m) ⇒ Ctor (KBiject m) KBProduct where
  selfmap f g = fw < − > bk where
    fw = Kleisli $
      λ(KBP x y) → (λ(x, y) → KBP x y) `liftM` mfuse ((runKleisli $ embed f) x, (runKleisli $ embed
```

```
    bk = Kleisli $
      λ(KBP x y) → (λ(x, y) → KBP x y) ‘liftM‘ mfuse ((runKleisli $ project f) x, (runKleisli $ project
instance (Monad m) ⇒ Absorbs (KBiject m) KBProduct KBOne where
  absorb = kbiject_product_absorb
  unabsorb = inv kbiject_product_absorb

kbiject_product_absorb :: Monad m ⇒ (KBiject m) (KBProduct KBOne 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) KBProduct where
  assoc = kbiject_product_assoc
  unassoc = inv kbiject_product_assoc

kbiject_product_assoc :: (Monad m) ⇒ (KBiject m) (KBProduct (KBProduct a b) c) (KBProduct a (KBP
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) KBProduct KBOne where

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

instance (Monad m) ⇒ CommunativeMonoidial (KBiject m) KBProduct KBOne where
```

3.4.3 Semiring Instance

```
instance (Monad m) ⇒ Annihilates (KBiject m) KBProduct KBZero where
  annihilates = (Kleisli $ λ(KBP ⊥ x) → return ⊥) < − > (Kleisli $ λx → return $ KBP x ⊥)

instance (Monad m) ⇒ Distributes (KBiject m) KBSum KBProduct where
  distribute = kbiject_distributes_iso
  undistribute = inv kbiject_distributes_iso

kbiject_distributes_iso :: (Monad m) ⇒ (KBiject m) (KBProduct (KBSum a b) c) (KBSum (KBProduct
kbiject_distributes_iso = 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) KBSum KBZero KBProduct KBOne where
```