Building a Better Arrow

Jonathan Fischoff

March 24, 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 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. Arrows assume that you can lift any function into the category you are writing code for, through the *arr* 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, and in fact overlap with less elegant solutions I had stumbled upon myself.

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.

I start with an umbrella term for both add and multiply.

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
module Data.Semiring where
import Prelude hiding ((.), id)
import Control.Category

class (Category cat) => Ctor cat constr | constr -> cat where
    selfmap :: cat a b -> cat c d -> cat (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.

```
promote :: (Category cat, Ctor cat constr) =>
           cat a b -> cat (constr a c) (constr b c)
```

```
promote = (flip selfmap) id

swap_promote :: (Category cat, Ctor cat constr) =>
                cat a b -> cat (constr c a) (constr c b)
swap_promote = selfmap id
```

It is probably not clear at this point but depending on the constructor 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.

$$a + 0 \leftrightarrow a$$

```
class (Category cat, Ctor cat op) =>
      Absorbs cat op id | op -> cat, op -> id where
    absorb   :: cat (op id a) a
    unabsorb :: cat a (op id a)
```

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

```
class (Category cat, Ctor cat op) =>
      Communative cat op | op -> cat where
    --also reflexive but an involution
    commute  :: cat (op a b) (op b a)
```

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

```
class (Category cat, Ctor cat op) =>
       Assocative cat op | op -> cat where
    assoc    :: cat (op (op a b) c) (op a (op b c))
    unassoc  :: cat (op a (op b c)) (op (op a b) c)
```

$$0 * a \leftrightarrow 0$$

```
class (Category cat, Ctor cat op) =>
      Annihilates cat op zero | op zero -> cat where
    annihilates   :: cat (op zero a) zero
    unannihilates :: cat zero (op zero a)
```

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

```
class (Category cat,
       Ctor cat add,
       Ctor cat multi) =>
       Distributes cat add multi | add multi -> cat where
    distribute   :: cat (multi (add a b) c) (add (multi a c) (multi b c))
    undistribute :: cat (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.

```
class (Category cat, Ctor cat dot,
       Assocative cat dot, Absorbs cat dot id) =>
       Monoidial cat dot id | dot id -> cat where
class (Monoidial cat op id, Communative cat op)  =>
        CommunativeMonoidial cat op id where

class (CommunativeMonoidial cat add zero,
       CommunativeMonoidial cat multi one,
       Annihilates cat multi zero, Distributes cat add multi) =>
       Semiring cat add zero multi one | add zero multi one -> cat where
```

From which I regain Arrow functionality by having the op be (,) or the Either.

```
first :: Semiring a add zero multi one =>
         a b c -> a (multi b d) (multi c d)
first = promote

second :: Semiring a add zero multi one =>
          a b c -> a (multi d b) (multi d c)
second = swap_promote

left :: Semiring a add zero multi one =>
        a b c -> a (add b d) (add c d)
left = promote

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 regained 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, choice uses sum types) for the monoid operator of communative monoid.

I'll create instances for the types in another post, assuming it is possible :).