Arrow Basics

Ted Cooper
theod@pdx.edu
CS510 – Spring 2014

1 Introduction

Here's a literate introduction to arrows and review of the basic Arrow typeclasses.

```
{-# LANGUAGE Arrows #-}
module ArrowBasics where

import Control.Arrow( Arrow, ArrowChoice, ArrowLoop, Kleisli(..)
                    , arr, first, second, (***), (&&&)
                    , left, right, (+++), (|||)
                    , loop )
import Control.Category(Category, (>>>), (.), id)
import Control.Monad(liftM)
import Data.Function(fix)
import Prelude hiding((.), id)

-- Count the occurrences of word w in a string
count  :: String → String
       → Int
count w = length ∘ filter (== w) ∘ words

-- > count "foo" "foo bar foo"
-- 2
```

In order to compose regular functions with functions that return types in a monad, we must lift them into the monad. This gets ugly quickly:

```
-- Count the occurrences of word w in a file and print the
-- count.
countFile  :: String → FilePath → IO ()
countFile w = (>>= print)
            ∘ liftM (count w)
            ∘ readFile

-- > countFile "foo" "fooBarFoo.txt"
-- 2
```

Haskell has an $Arrow$ typeclass that provides combinators we can apply to regular functions, monadic functions, and other things. We can use these combinators to rewrite $countFile$ if we can express the functions in its pipeline as values of types in the $Arrow$ typeclass:

```
countFileA  :: String → Kleisli IO FilePath ()
countFileA w = Kleisli readFile
           ⋙ arr (count w)
           ⋙ Kleisli print
```

This may not seem like much of an improvement, but we will shortly introduce more interesting arrow combinators.

What's up with this arr function and $Kleisli$ type constructor? Well, arr is like $liftM$ for arrows:

```
-- > :t arr
-- arr :: Arrow a ⇒ (b → c) → a b c
```

arr takes a function from b to c and lifts it into any arrow type from b to c.

2 Kleisli

We can use the $Kleisli$ newtype to make arrows out of functions of the type $Monad\, m \Rightarrow a \rightarrow m\, b$, which are also known as "Kleisli arrows":

```
-- > :i Kleisli
-- newtype Kleisli m a b
--   = Control.Arrow.Kleisli {Control.Arrow.runKleisli :: a → m b}
--         -- Defined in ‘Control.Arrow’

-- > :t readFile
-- readFile :: FilePath → IO String
-- > :t Kleisli readFile
-- Kleisli readFile :: Kleisli IO FilePath String
-- > :t print
-- print :: Show a ⇒ a → IO ()
-- > :t Kleisli print
-- Kleisli print :: Show a ⇒ Kleisli IO a ()
```

So Kleisli arrows have an embedded monad type, a parameter type, and a return type. If we match up the parameter and return types, we can compose arrows like composing functions. There are arrow composition operators:

($>>>$) composes left to right, i.e.
$appliedFirst >>> appliedSecond >>> appliedThird$

($<<<$) composes right to left, i.e.
$appliedThird <<< appliedSecond <<< appliedFirst$

3 Category

An aside on the $Category$ typeclass: Check out the type of ($>>>$):

```
-- > :t (⋙)
-- (⋙)
--   :: Control.Category.Category cat ⇒ cat a b → cat b c → cat a c
```

But isn't $(>>>)$ for composing arrows? Well, arrows are morphisms in a particular category, and $(>>>)$ composes category morphisms. What's a category? What's a morphism?

Adapted from `http://en.wikipedia.org/wiki/Category_(mathematics)`:

In short, a category is an algebraic structure with an identity "morphism" and associative composition of "morphisms". Things you need to form a category C:

- A class of objects $ob(C)$. A class can be a set like $\{1, 2, 3\}$ (called a "small class") or a collection that can't be represented as a set, like the class of sets that don't contain themselves (called a "proper class").
- A class of morphisms $hom(C)$. A morphism maps one object to one object in the category. We write the class of morphisms in a category from e.g. 1 to 2 as $hom(1, 2)$. For each object a there must be an identity morphism $id_a : a \to a$ that takes that object to itself. All morphisms must compose associatively, i.e. if $f \in hom(a, b)$ and $g \in hom(b, c)$ and $h \in hom(c, d)$, then $(f >>> g) >>> h = f >>> (g >>> h) \in hom(a, d)$

Let's look at the type of $(>>>)$ again:

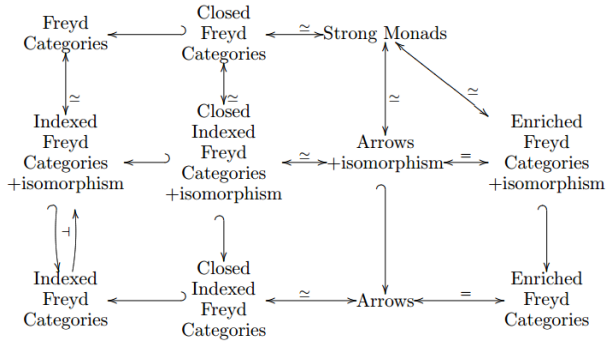
```
-- > :t (⋙)
-- (⋙)
--   :: Control.Category.Category cat ⇒ cat a b → cat b c → cat a c
```

So, it appears we should read this like $cat\ a\ b$ is a type representing morphisms from a to b, where a and b are objects in a category. According to **(author?)** [1], the fact that cat is a Haskell type, rather than a class of objects, is the key difference between Haskell Arrows and the morphisms in "Freyd Categories". Rather, arrows are morphisms in "Enriched Freyd Categories". Regardless, Haskell Arrows are morphisms in some category,

```
class Category a ⇒ Arrow a where ∘ .. (from Control.Arrow)
```

so we can certainly compose them using category morphism composition $(>>>)$.

Perhaps some of you will understand this (from **(author?)** [1]) better than I do:



4 (→) is an arrow

```
-- > :i (→)
-- ∘ ..
-- instance Category (→) -- Defined in ‘Control.Category’
```

```
-- ∘ ..
-- instance Arrow (→) -- Defined in ‘Control.Arrow’
```

Functions have an $Arrow$ instance, so we can use category and arrow combinators on them directly:

```
-- > head ⋙ ord $ "c"
-- 99
-- > ord ∘ head $ "c"
-- 99
```

However, if we'd like to use function arrows with another arrow type, we need to lift them into that arrow type with arr:

```
-- > let readRevA = Kleisli readFile ⋙ arr (init ⋙ reverse)
-- ∗ArrowBasics Control.Arrow Data.Function|
-- > readFile "fooBarFoo.txt" >>= (init ⋙ return)
-- "foo bar foo"
-- > runKleisli readRevA "fooBarFoo.txt"
-- "oof rab oof"
```

5 Arrow

The $Arrow$ typeclass provides a set of combinators in addition to arr, which let us do cool things with pairs (2-products):

```
-- > :i Arrow
-- class Category a ⇒ Arrow a where
--   arr :: (b → c) → a b c
--   first :: a b c → a (b, d) (c, d)
--   second :: a b c → a (d, b) (d, c)
--   (∗∗∗) :: a b c → a b’ c’ → a (b, b’) (c, c’)
--   (&&&) :: a b c → a b c’ → a b (c, c’)
--         -- Defined in ‘Control.Arrow’
```

Arrow combinators:

- $(***)$: Apply 2 arrows to the elements of a pair.

  ```
  -- > :t (∗∗∗)
  -- (∗∗∗) :: Arrow a ⇒ a b c → a b’ c’ → a (b, b’) (c, c’)
  ```

  Example:

  ```
  unitProduct :: Bool
  unitProduct  = ((+ 0) *** (++ "0") $ (1, "1")) == (1, "10")

  -- A simple way to apply a function to both elements
  -- of a homogeneously typed(?) pair.
  pairMap         :: (a → b) → (a, a) → (b, b)
  pairMap f (x, y) = (f x, f y)

  -- The same thing in point-free style, using arrow
  -- combinators
  pairMapA  :: (a → b) → (a, a) → (b, b)
  pairMapA f = f *** f
  ```

$Arrow$ instances must define at least 2 of these combinators: arr and $first$.

```
-- > :t arr
-- arr :: Arrow a ⇒ (b → c) → a b c
-- > :t first
-- first :: Arrow a ⇒ a b c → a (b, d) (c, d)
```

The rest are by default defined in terms of arr, $first$, and $(>>>)$. You may provide custom implementations to improve performance.

Arrow Laws (adapted from **(author?)** [4]):

Given

```
assoc :: ((a, b), c) → (a, (b, c))
assoc    ((x, y), z)  = (x, (y, z))
```

the following laws should hold for Arrow instances.

$arrid$ is identity

```
arr id ⋙ f = f ⋙ arr id = f
```

$(>>>)$ is associative

```
(f ⋙ g) ⋙ h = f ⋙ (g ⋙ h)
```

arr distributes across $(>>>)$:

```
arr (f ⋙ g) = arr (g ∘ f) = arr f ⋙ arr g
```

$first$ distributes across $(>>>)$:

```
first (f ⋙ g) = first f ⋙ first g
```

$first$ distributes(?) into function-land as $***id$

```
first (arr f) = arr (f *** id)
```

Arrows that affect different pair elements commute:

```
first f ⋙ arr (id *** g) = arr (id *** g) ⋙ first f
```

Applying a function to the first element of a pair then getting the first element of the result is the same as getting the first element of a pair then applying a function to the result:

```
first f ⋙ arr fst = arr fst ⋙ f
```

You can twirl ”left-associative” pair nests into ”right-associative” pair nests(?):

```
first (first f) ⋙ arr assoc = arr assoc ⋙ first f
```

5.1 Stream functions

```
newtype SF a b = SF { runSF :: [a] → [b] }

instance Category SF where
  id           = arr id
  SF f ∘ SF g  = SF (f ∘ g)

instance Arrow SF where
```

```
  arr f        = SF (map f)
  first (SF f) = SF (unzip ⋙ first f ⋙ uncurry zip)

first' (SF f) = SF $ (λxys → zip (f $ map fst xys) (map snd xys))

first'' (SF f) = SF $ map (λ(x, y) → (head $ f [x], y))

delay x = SF (init ∘ (x :))

-- arr works:
-- > runSF (arr (("(" ++) ⋙ (++ ")"))) ["a","b","c"]
-- ["(a)","(b)","(c)"]

-- first works:
-- > runSF (first $ arr (+1)) [(1,'a'),(2,'b')]
-- [(2,'a'),(3,'b')]

-- delay works:
-- > runSF (delay 0 &&& id) [1,2,3]
-- [(0,1),(1,2),(2,3)]
```

5.2 Joining computations

Consider a function that adds the result of two monadic computations:

```
addM :: (Monad m, Num a) ⇒ m a → m a → m a
f `addM` g = do x ← f
                y ← g
                return $ x + y

-- > return 3 `addM` return 4
-- 7
```

We can analogously create an arrow that adds the output of two arrows:

```
addA :: (Arrow a, Num c) ⇒ a b c → a b c → a b c
f `addA` g = f &&& g ⋙ arr (uncurry (+))

-- > const 3 `addA` const 4 $ undefined
-- 7
-- > (+ 3) `addA` (* 4) $ 2
-- 13
```

6 ArrowChoice

The *ArrowChoice* typeclass provides us with a set of additional combinators with *Eitherab,* a 2-element sum type.

```
-- > :i ArrowChoice
-- class Arrow a ⇒ ArrowChoice a where
--   left :: a b c → a (Either b d) (Either c d)
--   right :: a b c → a (Either d b) (Either d c)
```

```
--   (+++) :: a b c → a b' c' → a (Either b b') (Either c c')
--   (|||) :: a b d → a c d → a (Either b c) d
--         -- Defined in ‘Control.Arrow’
```

$(+++)$ transforms two arrows into an arrow from $Either$ to $Either$, much like how $(***)$ transforms two arrows into an arrow from pairs to pairs.

```
-- > ((+ 3) +++ (++ "3")) $ Left 1
-- Left 4
-- > ((+ 3) +++ (++ "3")) $ Right "1"
-- Right "13"
```

$(|||)$ transforms two arrows with different input types and the same output type to an arrow from $Either$ input type to that output type, allowing us to "join" values of $Either$ type wrapped in $Left$ and $Right$, respectively, to unwrapped outputs of the output type.

```
-- > (show ||| (++ "_was_already_a_string")) $ Right "asdf"
-- "asdf_was_already_a_string"
-- > (show ||| (++ "_was_already_a_string")) $ Left 1
-- "1"
```

$left$ lets us apply an arrow to $Lefts$ while passing through $Rights$ unchanged. It is analogous to $first$. Similarly, $right$ lets us apply an arrow to $Rights$ while passing through $Lefts$ unchanged. It is analogous to $second$.

```
-- > left (+1) $ Left 1
-- Left 2
-- > left (+1) $ Right "x"
-- Right "x"
--
-- > right (++"y") $ Left 1
-- Left 1
-- > right (++"y") $ Right "x"
-- Right "xy"
```

Just like we can compose arrows built with $first$ and $second$ to build arrows that operate independently on each element of a pair, we can compose arrows built with $left$ and $right$ to build arrows that operate independently on each type wrapped in an $Either$. Note that this is the same thing that $(+++)$ does. Indeed, $(+++)$ is defined in terms of $left$, $right$ (which is defined in terms of $left$), and $(>>>)$.

```
-- > left (+1) ⋙ right (++"y") $ Left 1
-- Left 2
-- > left (+1) ⋙ right (++"y") $ Right "x"
-- Right "xy"
--
-- > (+1) +++ (++"y") $ Left 1
-- Left 2
-- > (+1) +++ (++"y") $ Right "x"
-- Right "xy"

listcase []     = Left ()
listcase (x:xs) = Right (x,xs)
```

```
mapA  :: ArrowChoice a ⇒ a b c → a [b] [c]
mapA f = arr listcase
     ⋙ arr (const []) ||| (f *** mapA f ⋙ arr (uncurry (:)))

-- With function arrows:
-- > mapA (+1) [1,2,3]
-- [2,3,4]

-- With Kleisli arrows:
-- > runKleisli (mapA $ Kleisli print) [1,2,3]
-- 1
-- 2
-- 3
-- [(),(),()]

instance ArrowChoice SF where
  left (SF f) = SF (λxs → combine xs (f [y | Left y ← xs]))
    where combine (Left y : xs) (z:zs) = Left z : combine xs zs
          combine (Right y : xs) zs    = Right y : combine xs zs
          combine [] zs = []

-- > runSF (left (arr (+1))) $ map Left [1,2,3]
-- [Left 2,Left 3,Left 4]
-- > runSF (left (arr (+1))) $ map Right [1,2,3]
-- [Right 1,Right 2,Right 3]

-- View the input list as 3 parallel streams
-- (one for each element in the first list)
-- and delay each stream, moving elements ahead to ensure that
-- the stream of elements in position 0, stream of elements in
-- position 1, and stream of elements in position 2 appear in
-- the same order.
-- > runSF (mapA (delay 0)) [[1,2,3],[4,5],[6],[7,8],[9,10,11],[12,13,14,15]]
-- [[0,0,0],[1,2],[4],[6,5],[7,8,3],[9,10,11,0]]
```

7 ArrowLoop

We can model feedback loops with instances of the ArrowLoop class:

```
-- > :i ArrowLoop
-- class Arrow a ⇒ ArrowLoop a where
--   loop :: a (b, d) (c, d) → a b c
--         -- Defined in ‘Control.Arrow’
```

(author?) [3] : ”The reader who finds this argument difficult should work out the sequence of approximations in the call

```
runSF (loop (arr swap)) [1,2,3]
```

it is quite instructive to do so.”

Function instance:

```
instance ArrowLoop (→) where
    loop f b = let (c,d) = f (b,d) in c
```

This *loop* creates a recursive binding, much like one we'd use to write a recursive function using fix:

```
-- In case you don't remember fix, some implementations:
fix' f = let x = f x in x
fix'' f = f (fix' f)

repeat1 = fix (1:)

f (_, r) = (r, 1:r)

repeat1' = loop f undefined

-- Haskell will happily evaluate the recursive binding
-- forever, so we'd better limit the number of elements
-- we look at:

-- > take 10 repeat1
-- [1,1,1,1,1,1,1,1,1,1]
-- > take 10 repeat1'
-- [1,1,1,1,1,1,1,1,1,1]

repeatX x = fix (x:)

g (x, r) = (r, x:r)

repeatX' = loop g

-- > take 10 $ repeatX 'c'
-- "cccccccccc"
-- > take 10 $ repeatX' 'c'
-- "cccccccccc"

-- fib

fibGuyCurry = (λf → (λn → if n < 2 then n else f (n - 1) + f (n -2)))

fib = fix fibGuyCurry

fibGuyArrowInput (x, af) =
  ( af x
  , (λn → if n < 2 then n else af (n - 1) + af (n - 2)))

fib' = loop fibGuyArrowInput

-- > map fib [1..10]
-- [1,1,2,3,5,8,13,21,34,55]
-- > map fib' [1..10]
-- [1,1,2,3,5,8,13,21,34,55]
```

7.1 SF ArrowLoop instance

```
instance ArrowLoop SF where
  loop (SF f) = SF $ λas →
      let (bs, cs) = unzip (f (zip as (stream cs))) in bs
    where stream ~(x:xs) = x : stream xs

dup x = (x,x)

-- TODO: simple example
```

So what does this do? Each element in the input stream is zipped with an element from the feedback stream, the stream function is applied to that list of pairs, and the resulting list of pairs is unzipped into the output stream and the feedback stream. The marks an irrefutable pattern. Normally pattern matches are evaluated eagerly(right?) so Haskell can determine which branch to build a thunk proceeding down(?). If a pattern match is irrefutable, we can only have one branch, but we can defer evaluating the pattern match until the thunk is evaluated, i.e. for $stream$ when the value of an element in the list it returns is needed.

8 Arrows and feedback

Let's model synchronous digital signals as streams and logic gates as stream arrows! Check out $Circuits.hs$.

9 Future Work

- Rewrite TrivialParser.hs with PArrow or some version of the ParseArrow in Paterson.
- Toy XML Parser with HXT, maybe followed by FractalFlame example?

References

[1] Robert Atkey. What is a categorical model of arrows? *Electr. Notes Theor. Comput. Sci.*, 229(5):19–37, 2011.

[2] Jason Baker and luqui. How do i use fix, and how does it work?, 2011. [Online; accessed 28-April-2014].

[3] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129. Springer, 2005.

[4] Ross Paterson. A new notation for arrows. *SIGPLAN Not.*, 36(10):229–240, October 2001.