Arrows

Jonas Gehring

Universität Freiburg

January 21, 2008

Motivation - Ein einfaches Beispiel

Zählen der Vorkommen eines Wortes *w* in einem String:

```
count w = length . filter (==w) . words
```

Zählen der Vorkommen eines Wortes *w* in einer Datei:

```
count w = print .
      length . filter (==w) . words .
      readFile
```

Aufgrund der IO-Monaden in `print` und `readFile` nicht möglich:

```
readFile :: String -> IO String
print    :: Show a => a -> IO ()
```

Motivation - Ein einfaches Beispiel

`count` muss also nun Monaden verwenden:

```
count w = (>>= print) .
      liftM (length . filter (==w) . words) .
      readFile
```

Geht es auch übersichtlicher?

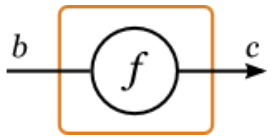
```
count w = readFile >>>
      arr words >>> arr (filter (==w)) >>>
      arr length >>> print
```

Arrows - Konzept

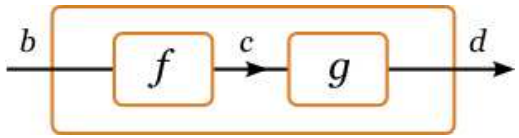
Definiere neue Klasse

```
class Arrow a where
    arr   :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
```

arr

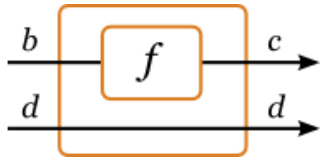


(>>>)

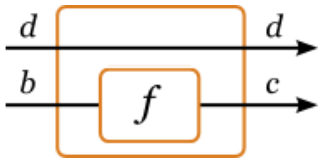


Arrows - Zusätzliche Funktionen

Um Arrows sinnvoll benutzten zu können, fehlen noch einige andere Funktionen:

- `first` 

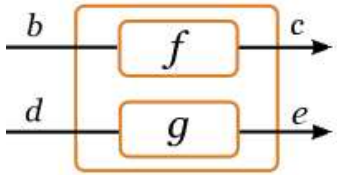

```
first :: a b c -> a (b,d) (c,d)
```

- `second` 


```
second :: a b c -> a (d,b) (d,c)
```

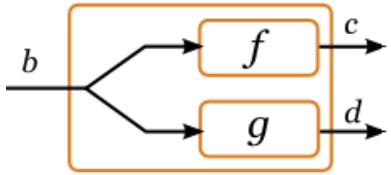
Arrows - Zusätzliche Funktionen

- `(***)` split




```
(***) :: a b c -> a d e -> a (b,d) (c,e)
```

- `(&&&)` fanout




```
(&&&) :: a b c -> a b d -> a b (c,d)
```

Arrows - Zusätzliche Funktionen

Die einzelnen Funktionen lassen sich gegenseitig definieren.
In der Prelude: `arr`, `(>>>)` und `first` als Primitiven.

```
class Arrow a where
  second f = arr swap >>> first f >>> arr swap
             where swap (x,y) = (y,x)
  f *** g  = first f >>> second g
  f &&& g  = arr (\b -> (b,b)) >>> f *** g
```

Arrows - Beispiel für Funktionen

Implementation eines Arrows für Funktionen:

```
instance Arrow (->) where
    arr f    = f
    f >>> g  = g . f
    first f  = f *** id
    f *** g ~(x,y) = (f x, g y)

doubleA = arr (\x -> x+x)

quadA = doubleA >>> doubleA

qaddA = (first quadA) >>> arr (\(x,y) -> x+y)
```

Arrows - Kompabilität

Arrows sind kompatibel zu Monaden, d.h. jede Monade kann mit einem Arrow modelliert werden.

```
newtype Kleisli m a b = K (a -> m b)

instance Monad m => Arrow (Kleisli m) where
    arr f       = K (\b -> return (f b))
    K f >>> K g = K (\b -> f b >>= g)
```

Anwendungen - Entscheidungen

Datentyp zur Symbolisierung des Berechnungsflusses

```
data Either a b = Left a | Right b
```

Neue Klasse `ArrowChoice`

```
class Arrow a => ArrowChoice a where
  left f  :: a b c -> a (Either b d) (Either c d)
  right f :: a b c -> a (Either d b) (Either d c)
  (+++)   :: a b c -> a d e ->
             a (Either b d) (Either c e)
  (|||)   :: a b d -> a c d ->
             a (Either b c) d
```

Anwendungen - Entscheidungen

Implementation von `ArrowChoice` für Funktionen:

```
instance ArrowChoice (->) where
    left f  = f +++ id
    right f = id +++ f
    f +++ g = (Left . f) ||| (Right . g)
    (|||)   = either
```

wobei `either` gegeben ist durch

```
either l r (Left x)  = l x
either l r (Right y) = r y
```

Anwendungen - Entscheidungen

Mit `ArrowChoice` kann man beispielsweise rekursive Berechnungen definieren

```
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 (:)))
```

Anwendungen - Streams

Streamfunktionen manipulieren einem Datenstrom, beispielsweise ein Array

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

Arrow-Instanz für SF:

```
instance Arrow SF where
    arr f          = SF (map f)
    SF f >>> SF g = SF (f >>> g)
    first (SF f)   = SF (unzip >>> first f
                        >>> uncurry zip)
```

Anwendungen - Streams

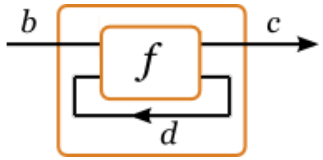
Streamfunkitonen eignen sich z.B. zur Modellierung elektrischer Schaltkreise

```
notA :: SF Bool Bool
notA =  arr (not)

orA :: SF (Bool,Bool) Bool
orA =  arr (\(x,y) -> x || y)

norA :: SF (Bool,Bool) Bool
norA =  orA >>> notA

delay   :: a -> SF a a
delay x =  SF (init . (x:))
```

Anwendungen - Streams

Einige Schaltungen (z.B. FlipFlops) benötigen ihre eigenen Ausgaben als Eingaben.

```
class Arrow a => ArrowLoop a where
    loop :: a (b,d) (c,d) -> a b c
```

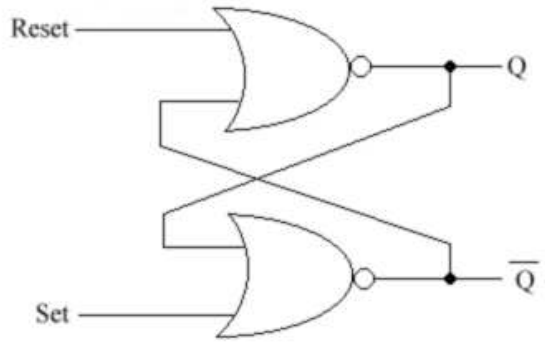



```
instance ArrowLoop SF where
    loop (SF f) = SF $ \bs ->
      let (cs,ds) =
        unzip (f (zip bs (stream ds))) in cs
      where stream ~(x:xs) = x: stream xs
```

Anwendungen - Streams

Mit `ArrowLoop` kann also nun ein FlipFlop modelliert werden

```
flipflop :: SF (Bool,Bool) (Bool,Bool)
flipflop =  loop (arr (\((reset,set),~(q,nq))
                  -> ((reset,nq),(set,q))) >>>
                 norA *** norA >>>
                 delay (True, False) >>>
                 arr id &&& arr id)
```



Zusammenfassung

Arrows bieten

- eine sinnvolle Alternative zu Monaden (und sind zu ihnen kompatibel)
- eine elegante Art, ”parallele” Berechnungen darzustellen
- anschauliche Operationen auf Berechnungsflüssen