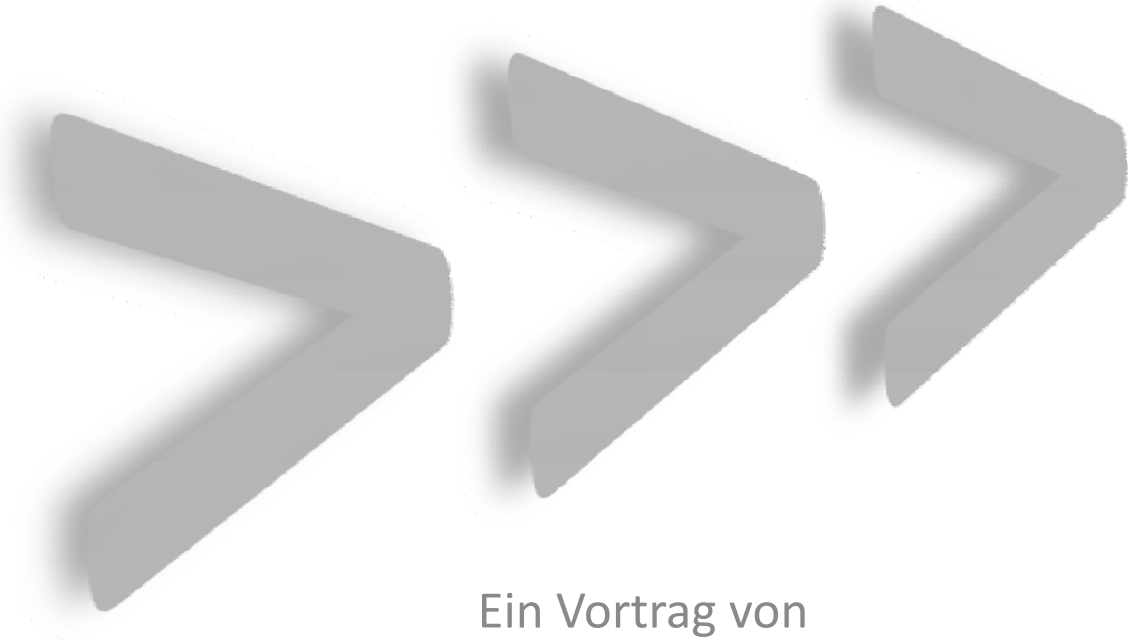
Arrows



Ein Vortrag von
Mario Rauschenberg, Bjarne Großmann

Agenda

- Einleitung
- Funktionen, Monaden und Arrows
- Arrows im Detail
- Beispiel

Einleitung

Was sind eigentlich „Arrows“?

Lediglich eine Verallgemeinerung der Monaden!

???

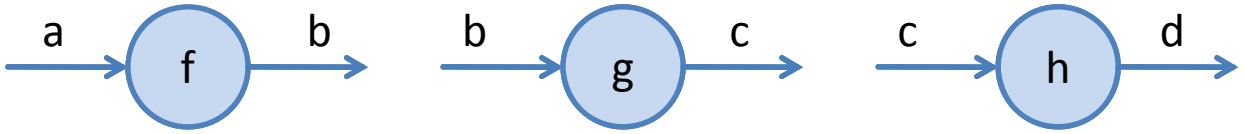
Einleitung

- Quizfrage: Was ist eine Monade?
  - a) Abstrakter Datentyp
  - b) Beschreibung einer einheitlichen Schnittstelle
  - c) Containerklasse für unterschiedliche Typen
  - d) Verpackte Berechnungen mit Seiteneffekten
  - e) Strategie, um Berechnungen zu kombinieren
  - f) Ich hab keine Ahnung

Back to the Roots

- Funktionen und ihre Komposition

```
f       :: a -> b
g       :: b -> c
h       :: c -> d
fgh     :: a -> d
fgh     = h . g . f
```



Back to the Roots

- Komplexere Funktionen

```
f1,g1,h1        :: a -> Maybe b
f2,g2,h2        :: a -> (b, s)
f3,g3,h3        :: a -> [b]
f4,g4,h4        :: a -> (s -> (a, s))

fgh = fn . gn . hn ???
```

Beispiel: Maybe

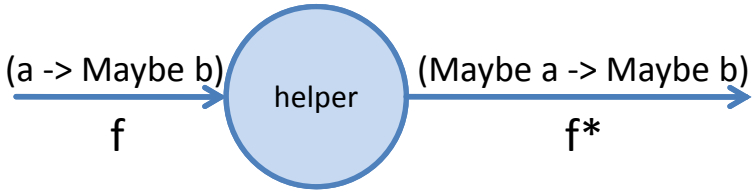
```
f   :: a -> Maybe b
g   :: b -> Maybe c
h   :: c -> Maybe d
fgh :: a -> Maybe d
```

1. Lösungsversuch

```
fgh x = case f x of
        Nothing -> Nothing
        Just y  -> case g y of
                     Nothing -> Nothing
                     Just z -> h z
```

Beispiel: Maybe

```
f    :: a -> Maybe b
g    :: b -> Maybe c
h    :: c -> Maybe d
fgh  :: a -> Maybe d
```



2. Lösungsversuch

```
helper                  :: (a -> Maybe b) -> (Maybe a -> Maybe b)
helper  f  Nothing      = Nothing
helper  f  (Just x)     = f x

fgh = (helper  h)  .  (helper  g)  .  f
```

Beispiel: Tupel

```
f               :: a -> (b, String)
g               :: b -> (c, String)
h               :: c -> (d, String)
fgh             :: a -> (d, String)

helper          :: (a -> (b, String)) -> ((a, String) -> (b, String))
helper g (fx, fs) = let (gx, gs) = g fx
                        in  (gx, fs++gs)

fgh   = (helper h) . (helper g) . f
```

Beispiel: Liste

```
f           :: a -> [b]
g           :: b -> [c]
h           :: c -> [d]
fgh         :: a -> [d]

helper      :: (a -> [b]) -> ([a] -> [b])
helper f x  = concat (map f x)

fgh = (helper h) . (helper g) . f
```

Beispiel: Status

```
f               :: a -> (s -> (b, s))
g               :: b -> (s -> (c, s))
h               :: c -> (s -> (d, s))
fgh             :: a -> (s -> (d, s))

helper          :: (a -> (s -> (b, s))) -> ((s -> (a, s)) -> (s -> (b, s)))
helper g x s    = let (x1, s1) = x s
                  in  g x1 s1

fgh = (helper h) . (helper g) . f
```

Beispiel: Zusammenfassung

```
f1          :: a -> Maybe b
helper1     :: (a -> Maybe b) -> (Maybe a -> Maybe b)
f2          :: a -> (b,  s)
helper2     :: (a -> (b,  s)) -> ((a,  s) -> (b,  s))
f3          :: a -> [b]
helper3     :: (a -> [b]) -> ([a] -> [b])
f4          :: a -> (s -> (d, s))
helper      :: (a -> (s -> (b, s))) -> ((s -> (a, s)) -> (s -> (b, s)))

fgh = (helper hn) . (helper gn) . fn
```

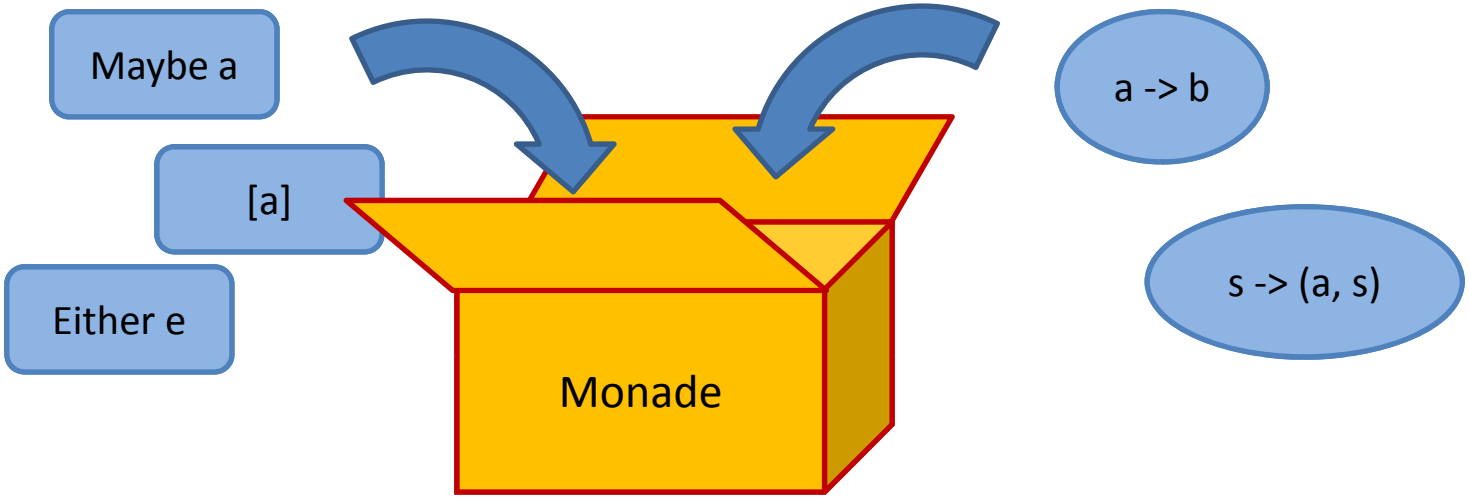
Monaden

```
newtype MyMonad t = M Maybe t
newtype MyMonad t = M (t, String)
newtype MyMonad t = M [t]
newtype MyMonad t = M (s -> (t, s))

f       :: a -> M b
helper  :: (a -> M b) -> (M a -> M b)
        :: (a -> M b) -> M a -> M b
        :: M a -> (a -> Mb) -> M b

(>>=)   = helper

fgh x   = f x >>= g >>= h
        ?=?  x >>= f >>= g >>= h
return  :: a -> M a

fgh x   = return x >>= f >>= g >>= h
```

Monaden

In Haskell:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```



Funktionskomposition

- Funktionen kombinieren

```
f :: a -> b                    g :: b -> c
```

  - Bisher

```
fg :: a -> c
fg = f . g
```

  - Funktionen als Parameter

```
comb          :: (a -> b) -> (b -> c) -> (a -> c)
comb h k      = k . h
```

  - (.) für einfache Funktionen

Funktionskomposition

- Funktionen kombinieren

```
f :: a -> m b      g :: b -> m c
```

  - Bisher

```
fg     :: a -> m c
fg x   = (f x) >>= g
```

  - Funktionen als Parameter

```
comb           :: (a -> m b) -> (b -> m c) -> (a -> m c)
comb h k x     = (h x) >>= k
```

  - (>>=) für monadische Funktionen?

```
(>>=) :: m a -> (a -> m b) -> m b
```

Funktionskomposition

- Abstraktion des Operators

```
comb                    :: (a -> m b) -> (b -> m c) -> (a -> m c)
newtype Kleisli m a b   = K { runK :: a -> m b }
(>->)                   :: Kleisli m a b -> Kleisli m b c -> Kleisli m a c
K f (>->) K g           = K (\x -> f x >>= g)
```

  - Der erste Arrow: Kleisli Arrow!
  - Anwendung ähnlich wie bei Monaden

```
f       :: K m a b
g       :: K m b c
fg x    = runK (f >-> g) x
```

Arrows

- Weitere Abstraktion
  - Kleisli Arrow nicht nur für Monaden:

```
f                       :: b -> c
newtype Arrow b c       = A ( runA :: b -> c)
(>>>)                   :: A b c -> A c d -> A b d
arr                     :: (b -> c) -> A b c
```

  - Arrows für beliebige Funktionen

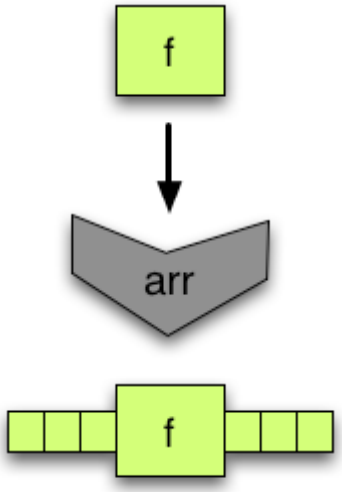
```
f       :: A a b
g       :: A b c
fg x    = runA (f >>> g) x
```

Was sind Arrows?

- Erklärung von Wikibooks
- Kombination von Funktions-Objekten (Roboter) mit verschiedenen In- und Outputs (Fließbänder)

arr

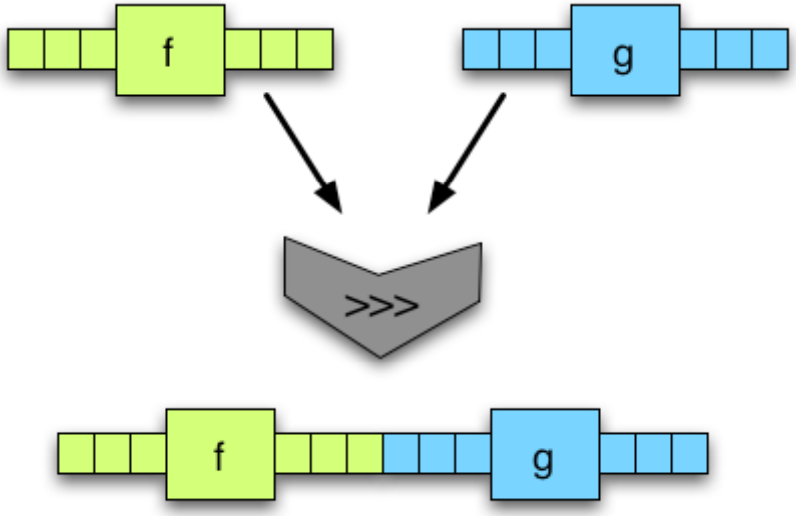
- Verknüpfung mit In- und Output



arr :: Arrow a => (b -> c) -> a b c

(>>>)

- Output von *f* als Input von *g*



(>>>) :: Arrow a => a b c -> a c d -> a b d

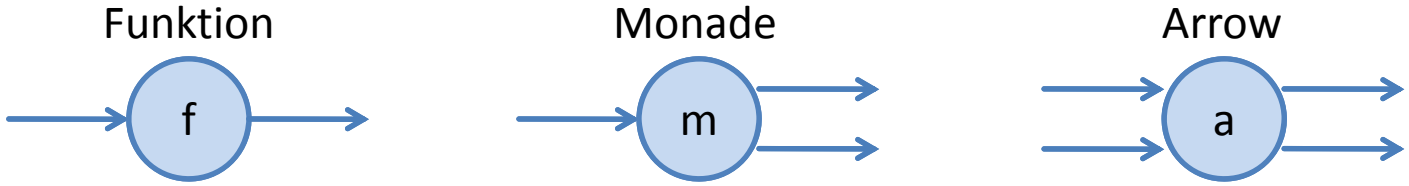
returnA

- Arrow – Identität

```
returnA :: Arrow a => a b b
returnA = arr id
```

Unterschied zu Monaden

- Arrows sind eine Erweiterung von Monaden
- Monaden sind eine Erweiterung von Funktionen
- Also: Arrows abstrahieren Funktionen
- Arrows haben expliziten Input



Monaden durch Arrows

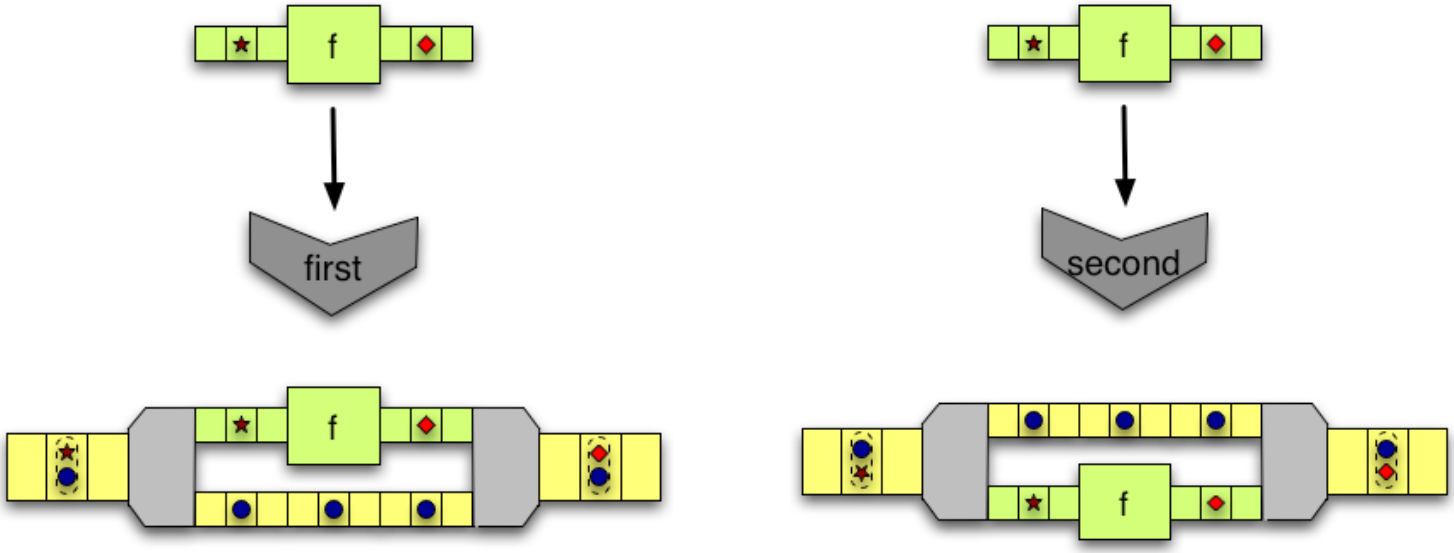
```
newtype Kleisli m a b = K (a -> m b)

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

„The real flexibility with arrows comes with the ones that aren't monads, otherwise it's just a clunkier syntax”

- *Philippa Cowderoy*

first & second

- Mehrere Inputs von denen einer verarbeitet wird.

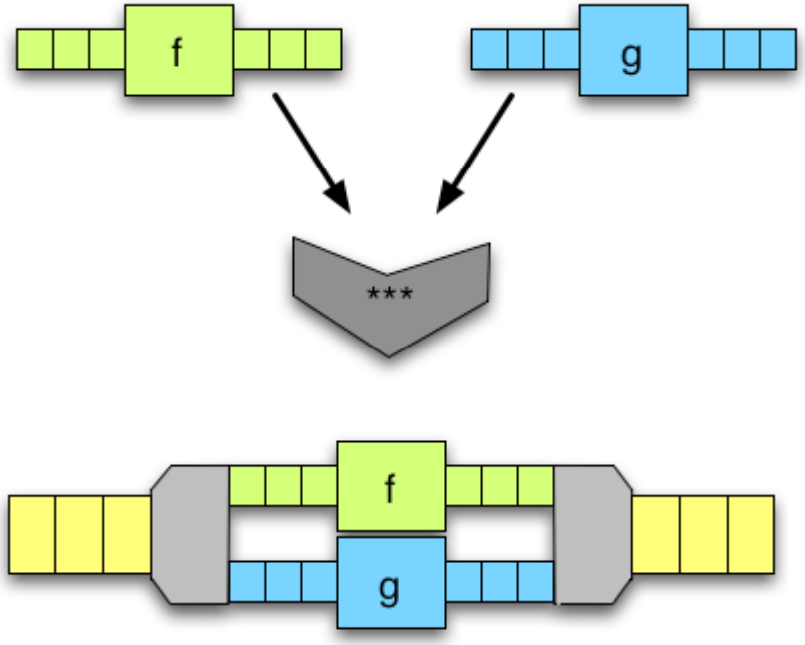


first :: Arrow a => a b c -> a (b, d) (c, d)

second :: Arrow a => a b c -> a (d, b) (d, c)

(***)

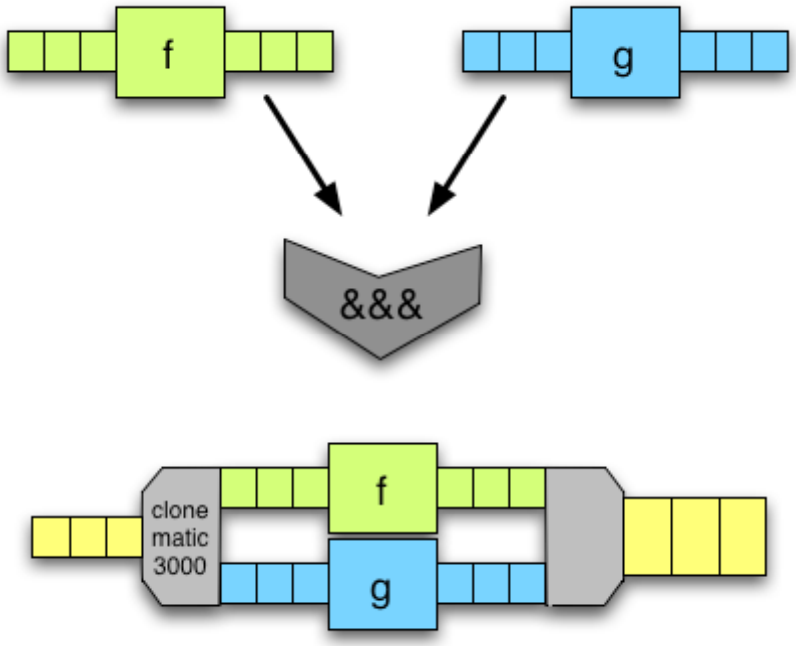
- Zwei Inputs durch jeweils anderes Funktionsobjekt bearbeitet



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

(&&&)

- Ein Input wird in beide Funktionen „geklont“.



(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)

Wozu das ganze?

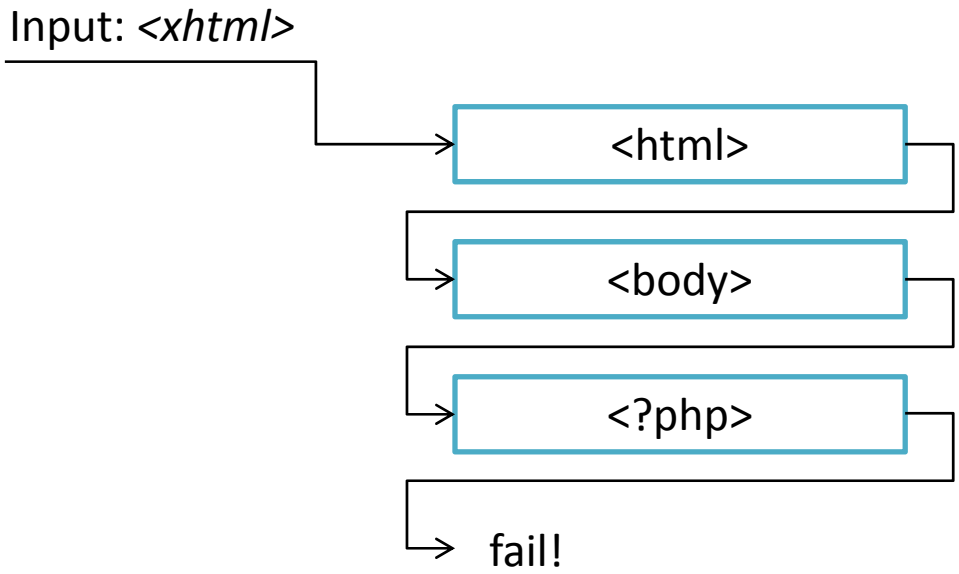
- Generalisierung von Monaden bzw. Funktionen und „Funktionsartigen“
- Größere Flexibilität als bei Monaden
- „Stream-Prinzip“

Funktionen als Arrows

```
instance Arrow (->) where
    arr f = f

    first f = f *** id

    second f = id *** f

    (***) f g ~(x,y) = (f x, g y)
```

Parser-Beispiel

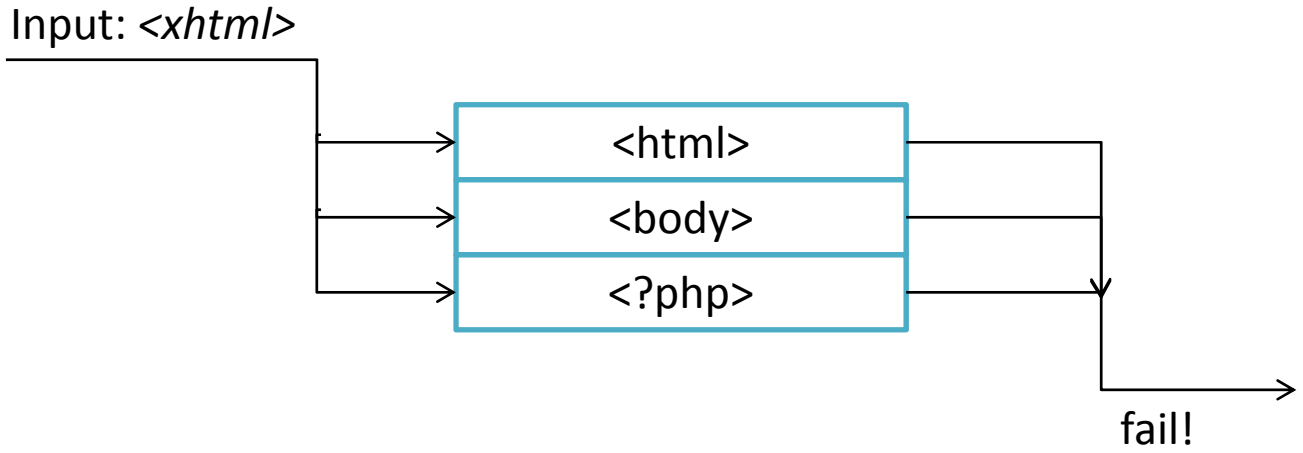
- Monadischer HTML-Parser



- Ineffizient!

Parser-Beispiel

- Erweiterung von Swierstra und Duponcheel: Vorausschauendes Parsen



Parser-Beispiel

- Monadischer Parser

```
newtype Parser s a = P ([s] -> Maybe (a,[s]))

symbol :: s -> Parsers s s
symbol s = P (\xs -> case xs of
                       [ ] -> nothing
                       (x : xs‘) -> if x == s
                                    then just (s, xs‘)
                                    else nothing )
```

Parser-Beispiel

- Monadische Parser kombinieren

```
instance MonadPlus Parsers where

P a (+++) P b = P (\s -> case a s of
                          Just (x, s') -> Just (x, s')
                          Nothing -> b s )
```

- Problem: „Space Leak“

Parser-Beispiel

- Erweiterung

```
data Parser s a b = P (StaticParser s) (DynamicParser s a b)

data StaticParser s = SP Bool [s]

newtype DynamicParser s a b = DP ( (a, [s]) -> (b, [s]) )

spChar :: Char -> StaticParser Char
spChar c = SP False [c]

dpChar :: Char -> DynamicParser Char Char Char
dpChar c = DP (\ ( _ , x:xs) -> (c, xs) )

simple :: Char -> Parser Char Char Char
simple c = P (spChar c ) (dpChar c)
```

Parser-Beispiel

```
Instance MonadPlus Parsers where
P (SP empty1 start1) DP (dp1) (+++) P (SP empty2 start2) DP (dp2) =
        P (
        SP (empty1 || empty2) (start1++start2)
        DP (\xss -> case xss of
            [ ] = if empty1 then dp1 [] else dp2 []
            j@(x:xs) =
                if x `in` start1 then dp1 j else
                if x `in` start2 then dp2 j else
                if empty1 then dp1 j else dp2 j
          )
        )
```

Parser-Beispiel

- Bind-Operator

```
(>>=) :: Parsers s a -> ( a -> Parsers s b) -> Parsers s b
```

- … ist nicht möglich, da der statische Teil von a „vergessen“ wird.

Parser-Beispiel

- Arrow-Parser

```
Instance Arrow (Parser s) where
arr :: Arrow a => (b -> s) -> a b s
arr  f = P  (SP True [ ]) (DP (\ (b,s) -> (f b, s) ) )

P (SP empty1 start1) DP (dp1) >>> P (SP empty2 start2) DP (dp2) =
            P (
            SP (empty1 && empty2)
                (start1 `union` if empty1 then start2 else [ ])
            DP (dp1 . dp2)
            )
```

Anwendungsbeispiele von Arrows

- Parser (z.B. auch HXT)
- Automaten
- Stream-Computing

Fazit

- Mächtiges Konzept für funktionsartige Vorgänge
- Allgemeinerer Entwurf als Monaden
- Weg von Pure-Functions
- Verständnis ist „nicht einfach“

Quellen (einige)

- http://www.haskell.org/haskellwiki
- http://www.wikibooks.org
- John Hughes: Generalising Monads to Arrows: http://www.cs.chalmers.se/~rjmh/Papers/arrows.pdf
- John Hughes: Programming with Arrows: http://www.cs.chalmers.se/~rjmh/afp-arrows.pdf
- Hudak, Courtney, Nilsson, Peterson: Arrows, Robots and Reactive Functional Programming http://www.haskell.org/yale/papers/oxford02/.oxford02.pdf
- Albert Lai: HXT Arrow Lessons: http://www.vex.net/~trebla/haskell/hxt-arrow/index.xhtml