Arrows in Haskell

Sven Hamer Martin Kartawijaya

Friday 15 January, 2010

Motivation

Wozu brauchen wir diese Arrows?

Definition:

Arrows are a new abstract view of computation. They serve much the same purpose as monads – providing a common structure for libraries – but are more general.

- Genereller als Monaden?
- Es gibt offenbar Berechnungen, die sich nicht mit Monaden ausdrücken lassen?
- Ein Beispiel: Ein monadischer Parser

Ein kleiner Rückblick

Mit Monaden lassen sich Berechnungen kapseln, welche einen Wert mit Typ a in eine Monade mit Typ b überführen.

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

Interface eines monadischen Parsers

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

class Monad m => MonadZero m where
  zero :: m a

class MonadZero => MonadPlus m where
  (++) :: m a -> m a -> m a
```

Implementierung eines monadischen Parsers

```
instance MonadZero Parser where
  zero = P (\s -> Nothing)

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

Problem mit dieser Implementierung

- Annahme: Grammatik mit Nicht-Terminal Symbolen
- Annahme: Der erste Parser schlägt fehl.
- Temporäre Daten `s` unnötig lange im Speicher für Backtracking:

```
... \s -> case a s of
             Just ...
             Nothing ...
```

- Space Leak!

Verfeinern des Parsers: Statischer Anteil

Idee: Aufteilen des Parsers in einen statischen und einen dynamischen Teil. (Annahme LL(1)-Parser)

Statischer Teil:

- First-Menge
- Prädikat: Akzeptiert leeren Input?

Dynamischer Teil:

- Funktion: Eingabewort nach gelesenes Token und Restwort

```
data StaticParser s = SP Bool [s]
newtype DynamicParser s a = DP ([s] -> (a, [s]))
data Parser s a = P (StaticParser s) (DynamicParser s a)
```

Neuimplementierung des Auswahl-Operators ++

```
instance MonadPlus Parser where
  P (SP empty1 first1) (DP p1) ++ P (SP empty2 first2) (DP p2) =
    P (SP (empty1 || empty2) (first1 ++ first2))
      (DP (\xs ->
            case xs of
              []       -> if empty1 then p1 [] else p2 []
              x : xs' -> if x `elem` first1 then p1 (x:xs') else
                         if x `elem` first2 then p2 (x:xs') else
                         if empty1 then p1 (x:xs') else p2 (x:xs')))
```

Neuimplementierung des Bind-Operators >>=

```
(>>=) :: Parser s a -> (a -> Parser s b) -> Parser s b
P (SP empty1 starters1) (DP p1) >>= f = ... ???
```

- Der zweite Parser ist hinter Funktion verborgen.
- Abhängig vom ersten Parser, somit ohne Ausführung desselben keine statischen Informationen erhältlich! :-(

Lösung des Problems

- Implementieren einer eigenen bind-Operation mit Parsern als Argument (keine funktionalen Abhängigkeiten mehr)
  Beispielsweise: `(<*>) :: Parser s (a->b) -> Parser s a -> Parser s b`
- Kosten: Verzicht des monadisches Interfaces
- Aber: Optimierungen dieser Art wünschenswert (Zugriff auf statische Informationen)
- **Deshalb:** Suche nach einer Verallgemeinerung der Monaden?

... If your application works fine with monads, you might as well stick with them. But if you're using a structure that's very like a monad, but isn't one, maybe it's an arrow.

Arrows im Vergleich zu Monaden

Grundlegende Operatoren analog zu den Monaden

Vergleich von Monaden und Arrows

Monade m

- Berechnung: liefert Wert Typ `a`.
- "Wert einpacken": `a -> m a`
- Das Ergebnis (exakter Wert) steht bereits fest, lediglich "auspacken".

Arrow a

- Berechnung: nimmt Wert Typ `b` und liefert Wert Typ `c`.
- "Funktion einpacken": `(b -> c) -> a b c`
- Das Ergebnis steht somit erst nach Anwendung eines Arguments fest.

Monade

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

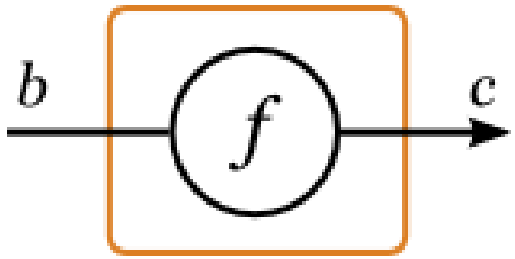


Abbildung:
arr-Operation

Arrow

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

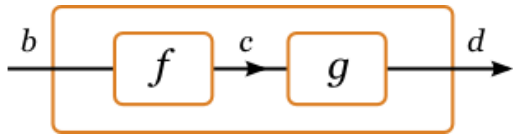


Abbildung:
>>>-Operation

Repräsentation von simplen Funktionen

Monade

```
data Func a = F {runF :: a}
  deriving Show

instance Monad Func where
  return     = F
  F a >>= f = (f a)
```

Arrow

```
instance Arrow (->) where
  arr    = id
  (>>>) = flip (.)
```

Kombinator: Ergebnisaddition

Monade

```
addM m1 m2 = m1 >>= \v1 ->
             m2 >>= \v2 ->
             return (v1 + v2)
```

Arrow

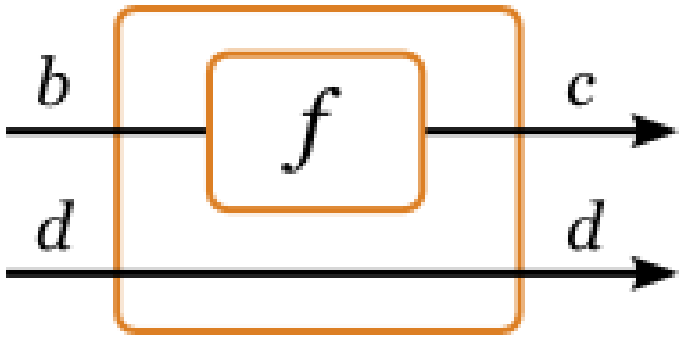
Bisher keine Möglichkeit für Zwischenspeicherung eines Wertes über eine weitere Berechnung (Arrow) hinweg.

Benötigt neue Operation!

Neue Operation: Zwischenspeicherung eines Wertes

```
class Arrow a where
  ...
  first :: a b c -> a (b, d) (c, d)
```

- Arrow konvertieren: Paare
- 1. Komponente führt Berechnung durch: `b -> c`
- 2. Komponente schleift Wert hindurch: `d -> d`



Monade

```
addM m1 m2 = m1 >>= \v1 ->
             m2 >>= \v2 ->
             return (v1 + v2)
```

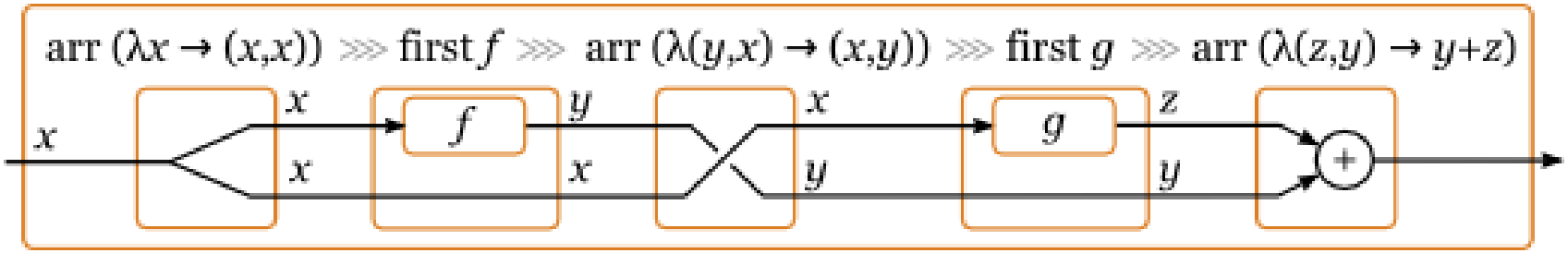
Arrow

Nun Implementierbar mittels `first`:

```
addA f g = arr (\ x -> (x, x)) >>>
           first f >>> arr (\ (y, x) -> (x, y)) >>>
           first g >>> arr (\ (z, y) -> y + z)
```

Arrow

```
addA f g = arr (\ x -> (x, x)) >>>
           first f >>> arr (\ (y, x) -> (x, y)) >>>
           first g >>> arr (\ (z, y) -> y + z)
```



Etwas unleserlich – Gabs da nicht diese do-Notation?

Bessere Lesbarkeit durch do-Notation

Monade

```
addM m1 m2 = do x <- m1
                y <- m2
                return x + y
```

Gibt es soetwas auch für Arrows? Ja! :-)

Arrow

```
addA f g = proc x -> do
               y <- f -< x
               z <- g -< x
               returnA -< y + z
```

Bessere Lesbarkeit durch do-Notation

Arrow

```
addA f g = proc x -> do
               y <- f -< x
               z <- g -< x
               returnA -< y + z
```

- `proc x` ist Analogie für eine Lambda-Funktion.
- `-<` "füllt" einen Wert in einen Arrow
- `<-` "liest" einen Wert aus einem Arrow
- `returnA` erzeugt einen trivialen Arrow.

Arrows sind Generalisierung von Monaden

Implementierung von Arrow durch Kleisli-Monade

```
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)
  first (K f) = K (\(b,d) -> f b >>= \c -> return (c,d))
```

Es gibt jedoch Arrows, die nicht durch Monaden implementiert werden können (siehe vorheriges Parser-Problem), somit sind Arrows eine echt Obermenge der Monaden.

Arrow-Kombinatoren

Nützliche Kombinatoren im Arrow-Interface

Für Arrows sind weitere nützliche Kombinatoren definiert:

- Analog zu `first`:

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

- Kombiniert zwei Arrows zu einen Arrow mit einem Paar als Ein- und Ausgabe, wobei jeweils eine Komponente der Ursprungsarrows entspricht:

```
(***) :: a b c -> a b' c' -> a (b, b') (c, c')
```

- Kombiniert zwei Arrows zu einem Arrow, welcher den Eingangswert auf ein Paar der Ergebnisse der beiden Ursprungsarrows abbildet:

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

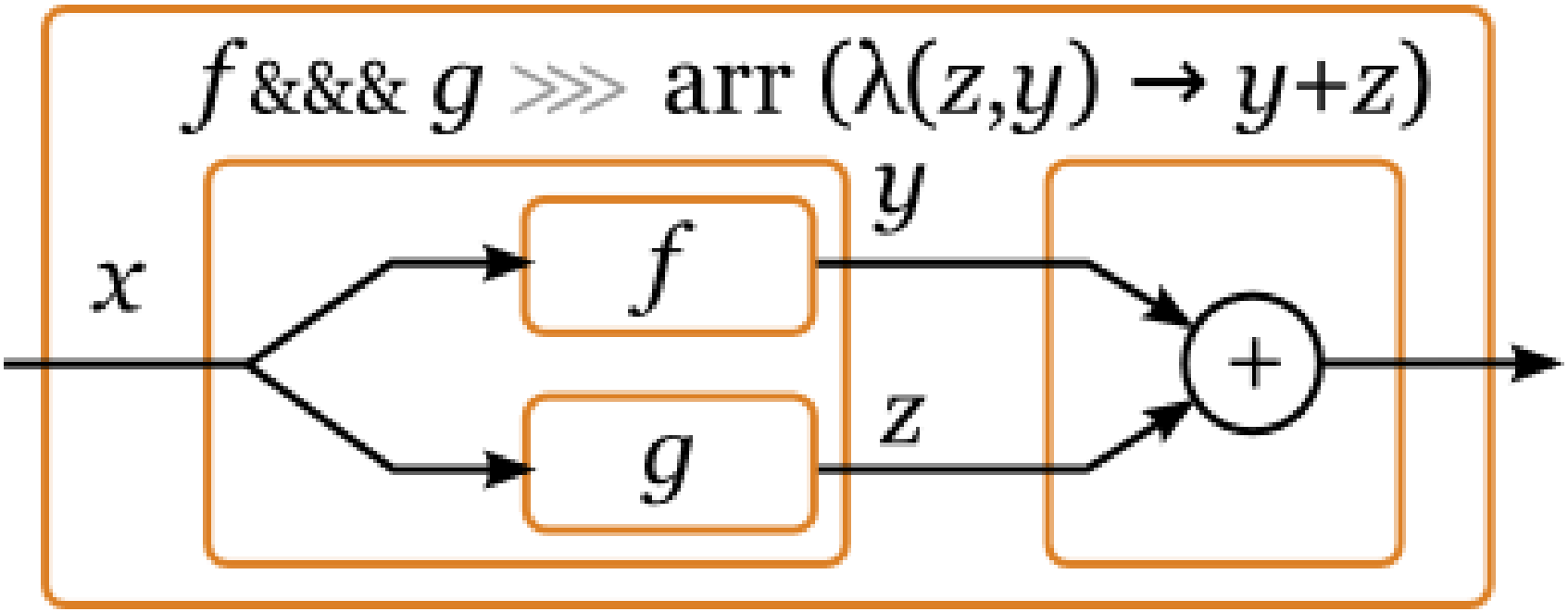
Standard-Implementierungen der Kombinatoren:

```
class Arrow a where
  ...
  second :: a b c -> a (d,b) (d,c)
  second f = arr swap >>> first f >>> arr swap
                  where   swap ~(x,y) = (y,x)

  (***) :: a b c -> a b' c' -> a (b,b') (c,c')
  f *** g = first f >>> second g

  (&&&) :: a b c -> a b c' -> a b (c,c')
  f &&& g = arr (\b -> (b,b)) >>> f *** g
```

Zeit für hübscheren Code?

Mit dem neuen `&&&`-Kombinator lässt sich die vorgestellte `addA`-Funktion auch ohne die do-Notation kürzer ausdrücken.

```
addA f g = f &&& g >>>
           arr (\(z,y) -> y+z)
```



Bedingte Auswertung von Arrows

Über die bisher verfügbaren Kombinatoren kann keine bedingte Auswertung implementiert werden, daher wird eine neue Klasse eingeführt.

```
class Arrow a => ArrowChoice a where
  left :: a b c -> a (Either b d) (Either c d)
```

Die Funktion `left` erzeugt einen neuen Arrow, der je nach Gestalt des Input den ursprünglichen Arrow ausführt oder den Eingabewert unverändert passieren lässt.

Beispiel für Funktionen

```
instance ArrowChoice (->) where
  left f = \x -> case x of
                   Left  b -> Left (f b)
                   Right d -> Right d
```

ArrowChoice: Zusätzliche Funktionen

```
right :: ArrowChoice a => a b c -> a (Either d b) (Either d c)
right a = arr mirror >>> left a >>> arr mirror
            where mirror (Left x)  = Right x
                  mirror (Right y) = Left y

(+++) :: ArrowChoice a =>
  a b c -> a b' c' -> a (Either b b') (Either c c')
f +++ g = left f >>> right g

(|||) :: ArrowChoice a => a b d -> a c d -> a (Either b c) d
f ||| g = (f +++ g) >>> arr untag
            where untag (Left x)  = x
                  untag (Right y) = y
```

Bedingte Auswertung von Arrows

Mit den nun verfügbaren Funktionen lässt sich ein `if-then-else` für Arrows implementieren.

```
test :: Arrow a => a b Bool -> a b (Either b b)
test a = (a &&& arr id) >>>
           arr (\(b, x) -> if b then Left x else Right x)

ifte :: ArrowChoice a => a b Bool -> a b c -> a b c -> a b c
ifte cond t e = test cond >>> (t ||| e)
```

Beispiele und Anwendungen

Verwendung und Einsatz in der wirklichen Welt

Stream Functions: Ein nichttrivialer Arrow

Stream Functions sind Arrows, die Listen verarbeiten. Dabei wird für jedes Element im Input genau ein Element im Output erzeugt.

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

instance Arrow StreamFunction where
  arr f = SF (map f)
  first (SF f) = SF (uncurry zip . first f . unzip)

instance ArrowChoice StreamFunction where
  left (SF f) = SF (\xs -> 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 = []
```

Arrows in the wild: Dokumentation und Projekte

- Viele auf Arrow basierende Projekte sind aus wissenschaftlichen Papers entstanden.
- Die meiste Dokumentation existiert daher nur in Form von wissenschaftlichen Arbeiten.

Beispiele

HXT	XML Verarbeitung
PArrows	Parser-Kombinator Bibliothek
...	Diverse Dataflow-Sprachen und GUI-Bibliotheken

Arrows in the wild: Arrows selbstgemacht

In Abhängigkeit vom verwendeten Compiler sind Arrows abweichend von der *reinen Lehre* zu Implementieren.

Arrows im GHC

```
data MyArrow a b = ...

instance Category MyArrow where
  id = ...
  MyArrow (...) . MyArrow (...) = MyArrow ...

instance Arrow MyArrow where
  arr (MyArrow (...)) = MyArrow ...
  first (MyArrow (...)) = MyArrow ...
```

Fazit

Wann Arrows, wann Monaden?

Fazit

- Arrows sind genereller als Monaden.
- Arrows ermöglichen Optimierung aufgrund statischer Information.
- Arrows bilden das Prinzip einer Berechnung besser als Monaden ab, da sie im Gegensatz zu diesen funktionsartig sind.
- Arrows sind aufwendiger zu implementieren als Monaden.
- Arrows ohne `proc`-Notation u.U. unhandlich.

Quellen

- http://www.haskell.org/arrows/
- http://en.wikibooks.org/wiki/Haskell
- John Hughes, Generalising Monads to Arrows, in Science of Computer Programming 37, pp67-111, May 2000
- John Hughes, Programming with Arrows, in 5th International Summer School on Advanced Functional Programming, LNCS 3622, pp 73-129, Springer, 2005

Vielen Dank für die Aufmerksamkeit

Gibt es Fragen?