Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 3: *Purity & noch mehr Typklassen*
Stand: 29. April 2016

Wiederholung:
Funktionen & Lazy Evaluation

Typklassen und Gesetze

Wir haben gelernt, dass wir mathematische Strukturen (wie z.B. Monoide) über *Typklassen* abbilden können.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a -- (<>) for infix usage
```

Typklassen und Gesetze

Wir haben gelernt, dass wir mathematische Strukturen (wie z.B. Monoide) über *Typklassen* abbilden können.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a -- (<>) for infix usage
```

Achtung: Es wird nicht vom Typsystem überprüft, ob die *Gesetze* der Typklasse (hier am Beispiel `Monoid`) eingehalten werden!

$$a \diamond e = a = e \diamond a$$

$$a \diamond (b \diamond c) = (a \diamond b) \diamond c$$

currying / partial application

Funktionen in `Haskell` sind normalerweise *curried*. Das erlaubt uns *partial application*, welche oft nützlich für die Praxis ist.

```
   foo_curried   :: a -> b -> c -> d
-- foo_curried   :: a -> (b -> (c -> d))
   foo_uncurried :: (a, b, c) -> d
```

currying / partial application

Funktionen in `Haskell` sind normalerweise *curried*. Das erlaubt uns *partial application*, welche oft nützlich für die Praxis ist.

```
   foo_curried   :: a -> b -> c -> d
-- foo_curried   :: a -> (b -> (c -> d))
   foo_uncurried :: (a, b, c) -> d
```

Wir lernen daraus, dass jede `Haskell`-Funktion *höchstens* einen Parameter annimmt. Sie gibt dann im Zweifelsfall eine Funktion mit einem Parameter weniger zurück, etc.

Wenn es dem Verständnis nicht im Wege steht, können wir allerdings auch weiter von Funktionen mit vielen Parametern sprechen.

Funktionsverkettung

Weil Funktionen in `Haskell` „first-class citizens" sind, können wir sie auch in sogenannten *Funktionen höherer Ordnung* anwenden (`map`, `filter`, ...).
Eine sehr nützliche higher-order function ist `(.)`, die Verkettung von zwei Funktionen noch Vorbild von $\circ$ in der Mathematik:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Funktionsverkettung

Weil Funktionen in `Haskell` „first-class citizens" sind, können wir sie auch in sogenannten *Funktionen höherer Ordnung* anwenden (`map`, `filter`, ...).
Eine sehr nützliche higher-order function ist `(.)`, die Verkettung von zwei Funktionen noch Vorbild von $\circ$ in der Mathematik:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Mit der Verkettung können wir Funktionen übersichtlicher schreiben, bis hin zum „pointfree style".

```
f, g :: (Num a) => a -> a
f x = incr (square x)
g   = incr . square
```

Lazy Evaluation

In Sprachen mit Lazy Evaluation wird vom RTS mit der Auswertung eines Ausdrucks (i.e. eines *thunks*) im Graphen gewartet, bis der Wert tatsächlich abgefragt wird. So wird verhindert, dass Ausdrücke unnötig (mehrfach) berechnet werden.

Lazy Evaluation

In Sprachen mit Lazy Evaluation wird vom RTS mit der Auswertung eines Ausdrucks (i.e. eines *thunks*) im Graphen gewartet, bis der Wert tatsächlich abgefragt wird. So wird verhindert, dass Ausdrücke unnötig (mehrfach) berechnet werden.

Lazy Evaluation erlaubt außerdem „unendliche“Datenstrukturen.

```
-- "infinite" list of fibonacci numbers
fibs :: [Integer]
fibs = 0 : 1 : zipWith (++) fibs (tail fibs)
```

Lazy Evaluation

In Sprachen mit Lazy Evaluation wird vom RTS mit der Auswertung eines Ausdrucks (i.e. eines *thunks*) im Graphen gewartet, bis der Wert tatsächlich abgefragt wird. So wird verhindert, dass Ausdrücke unnötig (mehrfach) berechnet werden.

Lazy Evaluation erlaubt außerdem „unendliche“Datenstrukturen.

```
-- "infinite" list of fibonacci numbers
fibs :: [Integer]
fibs = 0 : 1 : zipWith (++) fibs (tail fibs)
```

In Situationen, in denen Laziness nicht angebracht ist, kann sie in der Regel auch umgangen werden (z.B. mit `BangPatters` oder sogar eagerness-by-default).

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Warum sollte beim Programmieren auf Einhaltung von Typklassengesetzen geachtet werden?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Warum sollte beim Programmieren auf Einhaltung von Typklassengesetzen geachtet werden?
- Warum heißt es „pointfree style“ wenn wir doch öfter `(.)` verwenden?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Warum sollte beim Programmieren auf Einhaltung von Typklassengesetzen geachtet werden?
- Warum heißt es „pointfree style“ wenn wir doch öfter `(.)` verwenden?
- Gibt es zwischen den Signaturen `foo :: a -> b -> c` und `bar :: (a -> b) -> c` einen Unterschied?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Warum sollte beim Programmieren auf Einhaltung von Typklassengesetzen geachtet werden?
- Warum heißt es „pointfree style“ wenn wir doch öfter `(.)` verwenden?
- Gibt es zwischen den Signaturen `foo :: a -> b -> c` und `bar :: (a -> b) -> c` einen Unterschied?
- Was ist ein Vorteil von „unendlichen“ Datenstrukturen?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Warum sollte beim Programmieren auf Einhaltung von Typklassengesetzen geachtet werden?
- Warum heißt es „pointfree style“ wenn wir doch öfter `(.)` verwenden?
- Gibt es zwischen den Signaturen `foo :: a -> b -> c` und `bar :: (a -> b) -> c` einen Unterschied?
- Was ist ein Vorteil von „unendlichen“ Datenstrukturen?
- Was wäre eine Situation, in der Lazy Evaluation im Weg stehen kann?

Purity

referential transparency

Definition: Wir sagen, ein Ausdruck ist *referentially transparent* oder kurz *pur*, wenn wir ihn bei jedem Vorkommen durch seinen Rückgabewert ersetzen können, ohne dass sich das Verhalten des Programms ändert.

referential transparency

Definition: Wir sagen, ein Ausdruck ist *referentially transparent* oder kurz *pur*, wenn wir ihn bei jedem Vorkommen durch seinen Rückgabewert ersetzen können, ohne dass sich das Verhalten des Programms ändert.

Wir können zum Beispiel in `Haskell` jederzeit `(even 42)` durch `True` ersetzen, ändern tut sich dadurch nichts. Also ist der Ausdruck pur.
Ein weiterer purer Ausdruck ist `(filter (>0) [-1337..1337])`.

Seiteneffekte (I)

In der Mathematik sind alle Funktionen pur. Sobald wir jedoch programmieren, können Funktionen allerdings *Seiteneffekte* haben und dadurch ihre referential transparency verlieren.

Seiteneffekte (I)

In der Mathematik sind alle Funktionen pur. Sobald wir jedoch programmieren, können Funktionen allerdings *Seiteneffekte* haben und dadurch ihre referential transparency verlieren.

Definition: Wir sagen, dass ein Ausdruck einen *Seiteneffekt* (side effect) hat, wenn er einen Zustand (state) (ob global oder lokal) ändert oder Auswirkungen auf die Außenwelt hat (Dateien löschen, PC herunterfahren, . . .).

Seiteneffekte zu haben und referential transparency schließen sich gegenseitig aus!

Seiteneffekte (II)

Ein einfaches Beispiel aus der Programmiersprache Java:

```java
public static int fuenf()
{
    System.out.println("Seiteneffekt!");
    return 5;
}
```

Seiteneffekte (II)

Ein einfaches Beispiel aus der Programmiersprache Java:

```
public static int fuenf()
{
    System.out.println("Seiteneffekt!");
    return 5;
}
```

Die folgenden Ausdrücke haben somit alle eine unterschiedliche Bedeutung, auch wenn jedes Mal 10 zurück gegeben wird.

```
return 5 + 5;
return 5 + fuenf();
return fuenf() + fuenf();
```

Seiteneffekte (III)

In `Haskell` funktioniert das hingegen so nicht. Deswegen wird `Haskell` auch als eine *pure* Programmiersprache bezeichnet.

```
fuenf :: Int
fuenf = putStrLn "Seiteneffekt!" >> 5
```

Stattdessen kriegen wir einen Typfehler:

```
Couldn’t match expected type ‘Int’
            with actual type ‘IO b0’
In the expression: putStrLn "Seiteneffekt!" >> 5
```

xkcd by Randall Munroe, CC-BY-NC
https://xkcd.com/1312/

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, . . .).

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, . . .).

Für die Entwicklung von Software ist es ebenfalls oft hilfreich, sich auf unabhängige Teile des Codes zu konzentrieren, die nicht an einem globalen State hängen (Stichwort: Modularität).

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, ...).

Für die Entwicklung von Software ist es ebenfalls oft hilfreich, sich auf unabhängige Teile des Codes zu konzentrieren, die nicht an einem globalen State hängen (Stichwort: Modularität).

Zuletzt können einige mehr oder minder große Katastrophen verhindert werden, weil nicht einfach irgendwo Funktionen wie diese aufgerufen werden können.

```haskell
launchMissiles :: IO ()
launchMissiles = ... -- uh oh!
```

...and the bad

Allerdings ist eine Programmiersprache *komplett* ohne Seiteneffekte meist (mit ein paar wohldefinierten Ausnahmen) keine sehr nützliche Sprache.

Wenn wir keine Daten einlesen, keine Ergebnisse ausgeben und auch sonst nicht mit der Welt interagieren können, sind wir nicht mehr nützlich. Wir können zwar für uns interessante Werte berechnen (solange die Berechnung hardgecodet wurde), aber der Nutzer merkt davon nur, dass die CPU wärmer wird.

...and the bad

Allerdings ist eine Programmiersprache *komplett* ohne Seiteneffekte meist (mit ein paar wohldefinierten Ausnahmen) keine sehr nützliche Sprache.

Wenn wir keine Daten einlesen, keine Ergebnisse ausgeben und auch sonst nicht mit der Welt interagieren können, sind wir nicht mehr nützlich. Wir können zwar für uns interessante Werte berechnen (solange die Berechnung hardgecodet wurde), aber der Nutzer merkt davon nur, dass die CPU wärmer wird.

$\Rightarrow$ Wir brauchen Seiteneffekte, um in der Praxis einsetzbar zu sein.

`IO` in Haskell

Wenn wir in `Haskell` programmieren, müssen wir `IO`, einen speziellen Typen für diese Zwecke, verwenden. Zum Beispiel mit diesen Funktionen:

```
getLine  :: IO String        -- String einlesen
putStrLn :: String -> IO () -- String ausgeben
```

IO in Haskell

Wenn wir in Haskell programmieren, müssen wir IO, einen speziellen Typen für diese Zwecke, verwenden. Zum Beispiel mit diesen Funktionen:

```
getLine  :: IO String        -- String einlesen
putStrLn :: String -> IO () -- String ausgeben
```

Dabei ist zu beachten, dass es keine* Funktion mit dem Typen (IO a -> a) gibt. Wir sind also dazu gezwungen, Interaktionen mit der Außenwelt klar anzusagen und bestimmten Regeln zu folgen (dazu später mehr).

IO by example

Ein sehr simples Programm mit IO könnte so aussehen:

```haskell
-- All good! =)
main :: IO ()
main = do
  putStrLn "Wie ist Ihr Name?"
  name <- getLine
  -- name :: String
  -- Trotzdem noch im IO-Typen
  putStrLn $ "Hallo, " ++ name ++ "!"
```

`IO` by example

Ein sehr simples Programm mit `IO` könnte so aussehen:

```
-- All good! =)
main :: IO ()
main = do
  putStrLn "Wie ist Ihr Name?"
  name <- getLine
  -- name :: String
  -- Trotzdem noch im IO-Typen
  putStrLn $ "Hallo, " ++ name ++ "!"
```

Für einen einfachen Start gibt es in `Haskell` die `do`-Notation, die einen quasi-imperativen Programmierstil für den `IO`-Typen erlaubt. Mehr dazu in späteren Vorlesungen.

`IO` by bad example

Es gibt tatsächlich doch eine Funktion mit Typ `(IO a -> a)`, sie heißt `unsafePerformIO`. Der Name ist hier Programm.

Mit dieser Funktion lässt sich der `IO`-Typen umschiffen. Wir stellen aber fest, dass das oft nicht das ist, was wir *wollen*, weil wir so auch alle Vorteile von gekapseltem `IO` wieder verlieren.

IO by bad example

Es gibt tatsächlich doch eine Funktion mit Typ (`IO a -> a`), sie heißt `unsafePerformIO`. Der Name ist hier Programm.

Mit dieser Funktion lässt sich der `IO`-Typen umschiffen. Wir stellen aber fest, dass das oft nicht das ist, was wir *wollen*, weil wir so auch alle Vorteile von gekapseltem `IO` wieder verlieren.

```
-- You're doing it wrong! >.<
pureInput :: String
pureInput = unsafePerformIO . getLine
```

In Abgaben für diese Veranstaltung ist `unsafePerformIO` (sofern nicht explizit geregelt) an keiner Stelle im Code erlaubt.

Typklassen: `Functor`, `Applicative` und `Monad`

Functor

Definition: Functors

Wir haben bereits letzte Woche kurz über parametrisierte Typen wie `[]`, `Maybe` oder `Tree` gesprochen, die Instanzen für die Typklasse `Functor` haben.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Definition: Functors

Wir haben bereits letzte Woche kurz über parametrisierte Typen wie `[]`, `Maybe` oder `Tree` gesprochen, die Instanzen für die Typklasse `Functor` haben.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Oft hilft es, sich einen `Functor` als einen *Container* (z.B. eine Box) für seinen Parametertypen vorstellen. Dazu gleich noch mehr.

Funktorgesetze

Natürlich gibt es auch für `Functor` „ungeschriebene“ Gesetze, die alle Instanzen berücksichtigen sollten.

```
-- Instances should satisfy the following laws:
--  *  fmap id        ==  id
--  *  fmap (f . g)  ==  fmap f . fmap g
```

Funktorgesetze

Natürlich gibt es auch für `Functor` „ungeschriebene“ Gesetze, die alle Instanzen berücksichtigen sollten.

```
-- Instances should satisfy the following laws:
--  *  fmap id       ==  id
--  *  fmap (f . g)  ==  fmap f . fmap g
```

Diese Gesetze gehen sicher, dass die Anwendung von `fmap` nur die Werte im Container ändert, nicht die Struktur des Containers selbst.

Funktorgesetze

Natürlich gibt es auch für `Functor` „ungeschriebene“ Gesetze, die alle Instanzen berücksichtigen sollten.

```
-- Instances should satisfy the following laws:
--  *  fmap id       ==  id
--  *  fmap (f . g)  ==  fmap f . fmap g
```

Diese Gesetze gehen sicher, dass die Anwendung von `fmap` nur die Werte im Container ändert, nicht die Struktur des Containers selbst.

Fun Fact: Jeder Typ hat *höchstens* eine valide Funktorinstanz. Per Language-Extension kann der GHC auch Funktorinstanzen automatisch ableiten.

Instanzen

Zur Erinnerung hier auch nochmal die Instanzen zu `Functor` für `Maybe` und Listen:

```
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs
  -- or we could just say fmap = map
```

Instanzen

Zur Erinnerung hier auch nochmal die Instanzen zu `Functor` für `Maybe` und Listen:

```
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs
  -- or we could just say fmap = map


instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

Intuition

Meistens wird `Functor` als eine Klasse für *Container* vorgestellt und das ist auch in den meisten Fällen eine valide Sichtweise. Wir können die Typklasse allerdings auch interpretieren.

Intuition

Meistens wird `Functor` als eine Klasse für *Container* vorgestellt und das ist auch in den meisten Fällen eine valide Sichtweise. Wir können die Typklasse allerdings auch interpretieren.

Eine andere Sichtweise ist die eines Funktors als *Kontext* (z.B. für Berechnungen). So ändert zum Beispiel `fmap` eventuell den Wert, aber nicht den Kontext selbst. Später dazu noch mehr.

Intuition

Meistens wird `Functor` als eine Klasse für *Container* vorgestellt und das ist auch in den meisten Fällen eine valide Sichtweise. Wir können die Typklasse allerdings auch interpretieren.

Eine andere Sichtweise ist die eines Funktors als *Kontext* (z.B. für Berechnungen). So ändert zum Beispiel `fmap` eventuell den Wert, aber nicht den Kontext selbst. Später dazu noch mehr.

Schlussendlich ist aber ein `Functor` nicht mehr und nicht weniger als das, was seine *Definition* aussagt. Sich zu sehr auf eine Metapher zu versteifen, kann negative Auswirkungen auf den Lerneffekt haben.

Applicative

Definition: Applicatives

`Applicative` unterscheidet sich nicht stark von `Functor`. Wichtig sind eigentlich nur zwei Details:

Definition: Applicatives

`Applicative` unterscheidet sich nicht stark von `Functor`. Wichtig sind eigentlich nur zwei Details:

- Es gibt jetzt eine Funktion (genannt `pure`), um Dinge in den jeweiligen Kontext zu heben.

Definition: Applicatives

`Applicative` unterscheidet sich nicht stark von `Functor`. Wichtig sind eigentlich nur zwei Details:

- Es gibt jetzt eine Funktion (genannt `pure`), um Dinge in den jeweiligen Kontext zu heben.
- Bei Applicatives kann auch die *Funktion*, die angewendet werden soll, im Kontext stehen.

Definition: Applicatives

`Applicative` unterscheidet sich nicht stark von `Functor`. Wichtig sind eigentlich nur zwei Details:

- Es gibt jetzt eine Funktion (genannt `pure`), um Dinge in den jeweiligen Kontext zu heben.
- Bei Applicatives kann auch die *Funktion*, die angewendet werden soll, im Kontext stehen.

In `Haskell`-Code gegossen sieht das so aus:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative vs. Functor

Nebeneinander gestellt sind die Ähnlichkeiten von `fmap` (bzw. der Infixvariante `(<$>)`) und `(<*>)` sehr gut sichtbar:

```
(<$>) ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

Applicative vs. Functor

Nebeneinander gestellt sind die Ähnlichkeiten von `fmap` (bzw. der Infixvariante `(<$>)`) und `(<*>)` sehr gut sichtbar:

```
(<$>) ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

Viele Typen, die in `Functor` liegen, haben auch eine Instanz für `Applicative`. So zum Beispiel auch `Maybe`.

```
ghci>       even  <$> (Just 4) -- Just True
ghci> (Just even) <*> (Just 4) -- Just True
```

Intuition (I)

Vorgestellt wurde Applicative zuerst in einem Paper von McBride und Paterson mit dem Namen „Applicative Programming with Effects“.

`www.staff.city.ac.uk/~ross/papers/Applicative.html`

Intuition (I)

Vorgestellt wurde Applicative zuerst in einem Paper von McBride und Paterson mit dem Namen „Applicative Programming with Effects“.

`www.staff.city.ac.uk/~ross/papers/Applicative.html`

Der Name gibt hier erst Hinweise auf die beabsichtigte Intuition. Wenn wir uns an die Kontext-Interpretation erinnern, dann würde `pure` einen neutralen Kontext (ohne Effekte) für einen Wert erzeugen. Hingegen kann `(<*>)` auch auf den Kontext zugreifen und potentiell Effekte auf ihn haben.

Intuition (II)

Um etwas Gefühl für die Implikationen auf Codeebene zu bekommen, hier ein paar Beispiele:

```
ghci> (Just (+10)) <*> (Just 0)
Just 10
ghci> [(5+), (5*)] <*> [0,3,6]
[5,8,11,0,15,30]
```

Intuition (II)

Um etwas Gefühl für die Implikationen auf Codeebene zu bekommen, hier ein paar Beispiele:

```
ghci> (Just (+10)) <*> (Just 0)
Just 10
ghci> [(5+), (5*)] <*> [0,3,6]
[5,8,11,0,15,30]
```

Kontextlose Werte können direkt durch `pure` eingebettet werden:

```
ghci> [(+), (*)] <*> (pure 5) <*> [0,3,6]
[5,8,11,0,15,30]
```

Intuition (II)

Um etwas Gefühl für die Implikationen auf Codeebene zu bekommen, hier ein paar Beispiele:

```
ghci> (Just (+10)) <*> (Just 0)
Just 10
ghci> [(5+), (5*)] <*> [0,3,6]
[5,8,11,0,15,30]
```

Kontextlose Werte können direkt durch `pure` eingebettet werden:

```
ghci> [(+), (*)] <*> (pure 5) <*> [0,3,6]
[5,8,11,0,15,30]
```

Mit nur einer `Functor`-Instanz ist das nicht möglich:

```
ghci> fmap (+) (Just 5)
Just (5+) -- :: Maybe (Int -> Int)
ghci> fmap (Just (5+)) (Just 4)
-- ERROR!
```

Applicative Laws (I)

Natürlich gibt es auch Typklassengesetze für `Applicative`:

Applicative Laws (I)

Natürlich gibt es auch Typklassengesetze für `Applicative`:

- `pure id <*> v = v`

  Die Identitätsfunktion hat im Kontext keine Auswirkung.

Applicative Laws (I)

Natürlich gibt es auch Typklassengesetze für `Applicative`:

- `pure id <*> v = v`

  Die Identitätsfunktion hat im Kontext keine Auswirkung.

- `pure f <*> pure x = pure (f x)`

  Effektlose Funktion auf effektloses Argument angewendet ist das gleiche wie das Ergebnis direkt mit `pure` hochzuheben.

Applicative Laws (I)

Natürlich gibt es auch Typklassengesetze für `Applicative`:

- `pure id <*> v = v`

  Die Identitätsfunktion hat im Kontext keine Auswirkung.

- `pure f <*> pure x = pure (f x)`

  Effektlose Funktion auf effektloses Argument angewendet ist das gleiche wie das Ergebnis direkt mit `pure` hochzuheben.

- `u <*> pure y = pure ($ y) <*> u`

  Beim Anwenden einer effektvollen Funktion auf ein pures Argument spielt die Reihenfolge keine Rolle.

Applicative Laws (I)

Natürlich gibt es auch Typklassengesetze für `Applicative`:

- `pure id <*> v = v`

  Die Identitätsfunktion hat im Kontext keine Auswirkung.

- `pure f <*> pure x = pure (f x)`

  Effektlose Funktion auf effektloses Argument angewendet ist das gleiche wie das Ergebnis direkt mit `pure` hochzuheben.

- `u <*> pure y = pure ($ y) <*> u`

  Beim Anwenden einer effektvollen Funktion auf ein pures Argument spielt die Reihenfolge keine Rolle.

- `u <*> (v <*> w) = pure (.) <*> u <*> v <*> w`

  Assoziativität für `(<*>)`. Eventuell im Kopf typchecken.

Applicative Laws (II)

Es gibt auch ein Gesetz, das vorschreibt, wie sich ein `Applicative` zu seinem unterliegenden `Functor` verhalten soll:

```
g <$> x = pure g <*> x
```

Applicative Laws (II)

Es gibt auch ein Gesetz, das vorschreibt, wie sich ein `Applicative` zu seinem unterliegenden `Functor` verhalten soll:

```
g <$> x = pure g <*> x
```

Hier sehen wir auch, dass wir mit den Funktionen, die für `Applicative` definiert sein müssen auch jederzeit `fmap` implementieren können. Also ist jedes `Applicative` immer auch ein `Functor`, ob wir wollen oder nicht. Die Typklassenvoraussetzung zwingt uns nur zur Ehrlichkeit.

Instanzen

Wie bereits erwähnt ist `Maybe` auch ein `Applicative`. Der Standardkontext und die Implementation von `(<*>)` sind in diesem Fall recht offensichtlich:

```haskell
instance Applicative Maybe where
  pure             = Just
  Nothing  <*> _   = Nothing
  (Just f) <*> sth = f <$> sth -- Functor dependency
```

Instanzen

Wie bereits erwähnt ist `Maybe` auch ein `Applicative`. Der Standardkontext und die Implementation von `(<*>)` sind in diesem Fall recht offensichtlich:

```
instance Applicative Maybe where
  pure              = Just
  Nothing  <*> _    = Nothing
  (Just f) <*> sth = f <$> sth -- Functor dependency
```

Wir wissen, dass wir `fmap` bedenkenlos verwenden können, da die Typklasse eine Einschränkung auf Typen hat, die bereits in `Functor` liegen.

Monad

„Duck and Cover“, Public Domain
https://commons.wikimedia.org/wiki/File:DuckandCover.jpg

Was bisher geschah:

Was wir bisher in der Hierarchie der Typklassen gesehen haben:

Was bisher geschah:

Was wir bisher in der Hierarchie der Typklassen gesehen haben:

- `Functor` erlaubt uns, eine unverpackte Funktion auf einen verpackten Wert anzuwenden.

Was bisher geschah:

Was wir bisher in der Hierarchie der Typklassen gesehen haben:

- `Functor` erlaubt uns, eine unverpackte Funktion auf einen verpackten Wert anzuwenden.
- `Applicative` erlaubt uns, eine verpackte Funktion auf einen verpackten Wert anzuwenden.

Was bisher geschah:

Was wir bisher in der Hierarchie der Typklassen gesehen haben:

- `Functor` erlaubt uns, eine unverpackte Funktion auf einen verpackten Wert anzuwenden.
- `Applicative` erlaubt uns, eine verpackte Funktion auf einen verpackten Wert anzuwenden.

...aber wir haben noch keinen Weg, mit Funktionen umzugehen, die einen verpackten Wert zurück geben. Insbesondere nicht, wenn wir diese Funktion auf einen verpackten Wert anwenden wollen.

Was bisher geschah:

Was wir bisher in der Hierarchie der Typklassen gesehen haben:

- `Functor` erlaubt uns, eine unverpackte Funktion auf einen verpackten Wert anzuwenden.
- `Applicative` erlaubt uns, eine verpackte Funktion auf einen verpackten Wert anzuwenden.

... aber wir haben noch keinen Weg, mit Funktionen umzugehen, die einen verpackten Wert zurück geben. Insbesondere nicht, wenn wir diese Funktion auf einen verpackten Wert anwenden wollen.

Die Typklasse `Monad` liefert uns genau das mit der Funktion `(>>=)` (sprich: *bind*).

Example: Maybe

Sehen wir uns ein Beispiel an: Wie zu erwarten war ist `Maybe` auch eine Monade. Hier ist eine Funktion, die einen einfachen `Integer` erwartet und einen Integer im `Maybe`-Kontext zurück gibt:

```
half :: Integer -> Maybe Integer
half x = if even x then Just (x `div` 2)
                   else Nothing
```

Example: Maybe

Sehen wir uns ein Beispiel an: Wie zu erwarten war ist `Maybe` auch eine Monade. Hier ist eine Funktion, die einen einfachen `Integer` erwartet und einen Integer im `Maybe`-Kontext zurück gibt:

```
half :: Integer -> Maybe Integer
half x = if even x then Just (x `div` 2)
                    else Nothing
```

An diese Funktion können wir mithilfe von `(>>=)` auch eingepackt Werte übergeben:

```
ghci> Just 13  >>= half
Nothing
ghci> Just 128 >>= half >>= half >>= half
Just 16
ghci> Nothing  >>= half
Nothing
```

Definition: Monads

Hier sind die entsprechenden Typsignaturen:

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

Definition: Monads

Hier sind die entsprechenden Typsignaturen:

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

Die Funktion `return` ist außerordentlich unglücklich benannt. Es geht auch hier um das Heben eines Wertes in einen Kontext, *nicht* um einen Rückgabewert.

Definition: Monads

Hier sind die entsprechenden Typsignaturen:

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

Die Funktion `return` ist außerordentlich unglücklich benannt. Es geht auch hier um das Heben eines Wertes in einen Kontext, *nicht* um einen Rückgabewert.

Außerdem kann `return` in der Regel direkt die Implementation von `pure` übernehmen.
Dass `Monad` dafür eine direkte Funktion vorschreibt, ist ein Relikt aus alten Zeiten. Inhaltlich sind `return` und `pure` identisch.

Monad vs. Applicative vs. Functor

Ähnlich zu `Applicative` und `Functor` ergibt sich auch hier aus der Gegenüberstellung einiges an Erkenntnis:

```
(<$>) ::   (a ->   b) -> f a -> f b
(<*>) :: f (a ->   b) -> f a -> f b
(=<<) ::   (a -> m b) -> m a -> m b
-- (=<<) == flip (>>=)
```

Monad Laws

Auch diese Typklasse kommt natürlich mit ihren eigenen Gesetzen daher. Diese sind glücklicherweise etwas simpler als die von `Applicative`.

```
-- Left/Right identity & associativity
--
-- return a >>= f  == f a
-- m >>= return    == m
-- (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Intuition

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

Wir haben hier zwei Berechnungen (Werte in einem Kontext), nämlich `m a` und `m b`. Was `Monad` echt mächtiger macht als `Applicative` ist, dass `(>>=)` das *Ergebnis* der ersten Berechnung als Ausgangswert benutzen kann, um zu entscheiden, wie es weiter gehen soll.
So kommt auch die Idee ins Spiel, dass durch Monaden ein Begriff von Sequentialität entsteht (siehe `do`-Notation).

Intuition

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

Wir haben hier zwei Berechnungen (Werte in einem Kontext), nämlich `m a` und `m b`. Was `Monad` echt mächtiger macht als `Applicative` ist, dass `(>>=)` das *Ergebnis* der ersten Berechnung als Ausgangswert benutzen kann, um zu entscheiden, wie es weiter gehen soll.
So kommt auch die Idee ins Spiel, dass durch Monaden ein Begriff von Sequentialität entsteht (siehe `do`-Notation).

Es gibt auch eine schwächere Version von `(>>=)`, nämlich `(>>)` (sprich: *shove*), mit der Signatur `(>>):: m a -> m b -> m b` . Diese ignoriert das *Ergebnis* von der ersten Berechnung, allerdings nicht ihre *Effekte*.

Alternative Monaden

Monaden können noch auf eine andere Art und Weise definiert werden, die näher an den mathematischen Grundlagen im Feld der Kategorientheorie (dazu später mehr) liegen.
Diese andere Interpretation würde in Code etwa so aussehen:

```
class Applicative m => Monad m where
  unit  :: a -> m a
  join  :: m (m a) -> m a
--(>>=) :: m a -> (a -> m b) -> m b
```

Alternative Monaden

Monaden können noch auf eine andere Art und Weise definiert werden, die näher an den mathematischen Grundlagen im Feld der Kategorientheorie (dazu später mehr) liegen.
Diese andere Interpretation würde in Code etwa so aussehen:

```
class Applicative m => Monad m where
  unit  :: a -> m a
  join  :: m (m a) -> m a
--(>>=) :: m a -> (a -> m b) -> m b
```

Diese zwei Sichtweisen auf die Definition von Monaden sind dahingehend äquivalent, dass sich `(>>=)` durch `join` ausdrücken lässt und umgekehrt.

```
m >>= g  = join (g <$> m)
join x   = x >>= id
```

Instanzen

Die Monadeninstanz für `Maybe` erinnert an die gleiche Instanz für `Applicative`, aber im Fall des nicht-leeren Kontextes sind sie Unterschiedlich.

```
instance Monad Maybe where
  return         = Just
  Nothing  >>= _ = Nothing
  (Just x) >>= g = g x
```

Instanzen

Die Monadeninstanz für `Maybe` erinnert an die gleiche Instanz für `Applicative`, aber im Fall des nicht-leeren Kontextes sind sie Unterschiedlich.

```
instance Monad Maybe where
  return         = Just
  Nothing  >>= _ = Nothing
  (Just x) >>= g = g x
```

Die *wirklich* interessanten Monaden (z.B. `State`, Reader, ...) werden wir in einer späteren Vorlesung noch Behandeln. Aber was wäre die Instanz für `[]`?

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

- `Haskell` stellt Monaden in den Vordergrund, indem es `IO` mit einer magischen Monade behandelt.

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

- `Haskell` stellt Monaden in den Vordergrund, indem es `IO` mit einer magischen Monade behandelt.
- `Haskell` stellt außerdem für alle Monaden speziellen syntaktischen Zucker in Form der `do`-Notation bereit.

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

- `Haskell` stellt Monaden in den Vordergrund, indem es `IO` mit einer magischen Monade behandelt.
- `Haskell` stellt außerdem für alle Monaden speziellen syntaktischen Zucker in Form der `do`-Notation bereit.
- `Monad` als Abstraktion gibt es schon länger als Alternativen wie `Applicative` oder `Arrow`.

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

- `Haskell` stellt Monaden in den Vordergrund, indem es `IO` mit einer magischen Monade behandelt.
- `Haskell` stellt außerdem für alle Monaden speziellen syntaktischen Zucker in Form der `do`-Notation bereit.
- `Monad` als Abstraktion gibt es schon länger als Alternativen wie `Applicative` oder `Arrow`.
- Je mehr Tutorials zu Monaden geschrieben werden, desto mehr Leute denken, Monaden wären ein sehr komplexes Thema und desto mehr Tutorials werden geschrieben.

Special Snowflakes

Was macht Monaden in Haskell so besonders, dass alle andauernd über sie reden wollen?

- `Haskell` stellt Monaden in den Vordergrund, indem es `IO` mit einer magischen Monade behandelt.
- `Haskell` stellt außerdem für alle Monaden speziellen syntaktischen Zucker in Form der `do`-Notation bereit.
- `Monad` als Abstraktion gibt es schon länger als Alternativen wie `Applicative` oder `Arrow`.
- Je mehr Tutorials zu Monaden geschrieben werden, desto mehr Leute denken, Monaden wären ein sehr komplexes Thema und desto mehr Tutorials werden geschrieben.

Für den Moment kommen wir einfach nicht drum herum...

Die Hierarchie

Es gibt natürlich noch mehr Typklassen die nützliche Strukturen (ob aus der Mathematik oder nicht) abbilden…

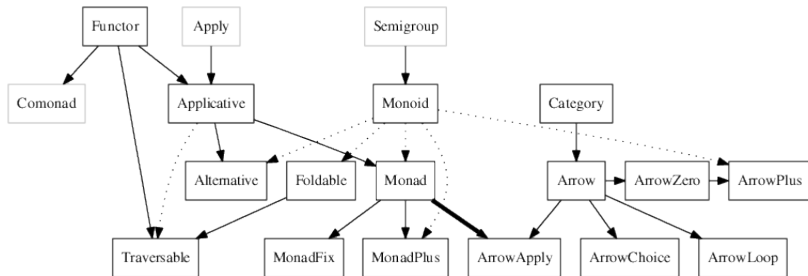


Diagramm zu Typklassen aus der *Typeclassopedia* im `Haskell`-Wiki
`https://wiki.haskell.org/File:Typeclassopedia-diagram.png`

further reading

Falls noch einige Konzepte unklar geblieben sind, helfen vielleicht diese Quellen weiter:

further reading

Falls noch einige Konzepte unklar geblieben sind, helfen vielleicht diese Quellen weiter:

- **Functors, Applicates and Monads in pictures**
  Gut aufgearbeitet und leicht verständlich, hat schon vielen Leuten über den Berg geholfen.
  `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

further reading

Falls noch einige Konzepte unklar geblieben sind, helfen vielleicht diese Quellen weiter:

- **Functors, Applicates and Monads in pictures**
  Gut aufgearbeitet und leicht verständlich, hat schon vielen Leuten über den Berg geholfen.
  `http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

- **The Typeclassopedia**
  Ein bekannter und bewährter Artikel, an dem kaum jemand auf dem Weg zu `Haskell` vorbei kommt; erklärt viele Typklassen nochmal im Detail und im Kontext.
  `https://wiki.haskell.org/Typeclassopedia`

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Purity
  - Definitionen von Purity und Seiteneffekt
  - `IO`-Typ und Motivation
  - ~~unsafePerformIO~~

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Purity
  - Definitionen von Purity und Seiteneffekt
  - `IO`-Typ und Motivation
  - ~~unsafePerformIO~~
- Typklassen
  - `Functor` (`fmap`)
  - `Applicative` (`pure, <*>`)
  - `Monad` (`return, >>=`)

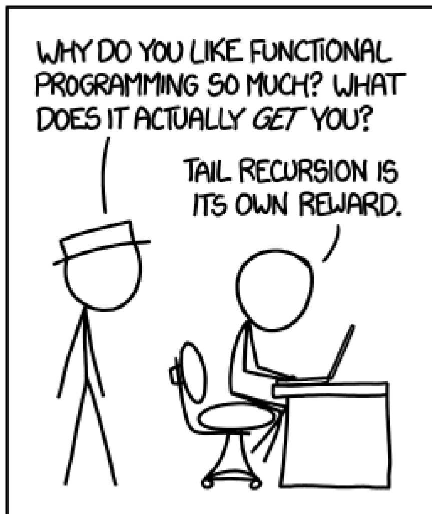
Ausblick

Vorschau: Was machen wir nächste Woche?

- Wiederholung: Vorlesung 3
- Parsing in `Haskell`
  Wie kommen wir von `String` zu sinnvollen Datentypen?

  (Am Beispiel von `AttoParsec`)

Fragen?



xkcd by Randall Munroe, CC-BY-NC
`https://xkcd.com/1270/`