Fortgeschrittene Funktionale Programmierung

Teil 5: Records, Linsen, Template Haskell

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

11. Dezember 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 5: Records, Linsen, Template Haskell

1. Records
   - GHC Spracherweiterungen für Records
   - Zusammenfassung Records
2. TemplateHaskell
   - QuasiQuotes
   - Spleißen
   - Beispiele
3. Funktionale Referenzen (Linsen)
   - Problematik
   - Rank-N Types
   - van Laarhoven Linsen
   - Traversal
   - Linsen Hierarchy
   - Prismen
   - Iso

Record-Syntax

Konstruktoren mit vielen Typ-gleichen Argumenten werden schnell unübersichtlich:

```
data Person' = Person'  String Int Int Int Int
p0 = Person' "Tyrion" 135 26 7 0
```

Record-Syntax erlaubt es, die *Argumente eines Konstruktors* zu benennen; diese werden dann auch als **Felder** bezeichnet.

```
data Person  = Person { name::String, height,age,mates,offspring::Int }
p2 = Person  {height=166, age=35, offspring=3, mates=3, name="Cersei"}
p3 = Person  "Jaimie" 187 35 1 3      -- alte Syntax auch noch erlaubt
```

- Reihenfolge der Felder innerhalb geschweifter Klammern ist immer beliebig
- Datentypen können nachträglich zu Records gemacht werden: Werden keine geschweiften Klammern verwendet, gilt die Reihenfolge wie in der Definition, wie üblich.

Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } | n > 0 = True
hasChildren _ = False

> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. der Mustervergleich betrifft nur einen Teil der Felder.

Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } = n > 0
```

```
> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. der Mustervergleich betrifft nur einen Teil der Felder.

Record Projektionen

Projektionen (engl. selector functions) werden für jeden Feldnamen automatisch definiert:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
> :t name
name :: Person -> String

> name p3
"Jaimie"

> :t age
age :: Person -> Int

> age p3
35
```

Projektionen und newtype

Die automatisch definierten Projektionen nutzt man gerne bei newtype Deklarationen aus, da in diesem Fall eine Projektion immer nützlich ist, um den Wrapper-Konstruktor zu entfernen:

```
newtype ZipList a = ZipList { getZipList :: [a] }

> :t getZipList
getZipList :: ZipList a -> [a]


> getZipList $ ZipList [(*2),(+2)] <*> ZipList [5,7]
[10,9]

> ZipList [(*2),(+2)] <*> ZipList [5,7]
ZipList {getZipList = [10,9]}
```

Record Pseudoupdate

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
p1 = Person "Tyrion" 135 26 7 0
```

Funktionale "Field-updates" sind ebenfalls möglich. Dabei werden natürlich Kopien erstellt — denn bestehende Werte werden in der funktionalen Welt ja nie verändert!

```
p4 = p1 { name = "Imp" }
p5 = p1 { mates = 2 + mates p1 }

> p5
Person {name = "Tyrion", height = 135, age = 26,
        mates = 9, offspring = 0}
> p1
Person {name = "Tyrion", height = 135, age = 26,
        mates = 7, offspring = 0}
```

Record Refactoring

Records kann man bei Bedarf nachträglich leicht erweitern:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

p6 = Person { name="Daenerys", height=157, offspring=0 }
```

ist gültiger Code (wenn z.B. `age` und `mates` nachträglich in die Definition aufgenommen wurden). Es wird interpretiert als:

```
p6 = Person "Daenerys" 157 undefined undefined 0
```

GHC kann in diesem Fall Warnungen ausgeben, dass nicht alle Felder eines Records initialisiert wurden.

Diese Warnungen sollte man nach einer solchen nachträglich Erweiterung des Record-Typs aber auch abarbeiten!

Standardwerte für Records definieren

Um die Wartbarkeit zu erhöhen ist es manchmal ratsam, einen globalen Record mit Default-Werten anzulegen:

```
data Bar = Bar { bar :: Int, baz :: Int, quux :: Int }

barDefault = Bar { bar = 1, baz = 2, quux = 3 }

myBar = barDefault { bar = 69, baz = 42 } -- quux == 3
```

Ansatt einen neuen Record anzulegen, wird lediglich ein Field-Update des Standardwerts durchgeführt

Fügt man später ein weiteres Feld ein, muss man lediglich den Default-Wert anpassen.
Nachteil: Der Kompiler gibt dann keine Warnungen mehr heraus, wenn nicht alle Felder *korrekt* initialisiert werden.

Records und Summentypen

Record-Syntax kann auch mit Alternativen verwendet werden:

```
data UniP = Student { name::String, matrikel::Int }
          | Dozent  { name::String, titel::String }

u1 = Dozent  { name="Steffen", titel="Dr" }
u2 = Student { name="Max",  matrikel=666  }

isDozent :: UniP -> Bool
isDozent Dozent {} = True
isDozent    _      = False

isSteffen :: UniP -> Bool
isSteffen p = "Steffen" == name p
```

Partielle Projektionen

Record-Syntax kann auch mit Alternativen verwendet werden:

```
data UniP = Student { name::String, matrikel::Int }
          | Dozent  { name::String, titel::String }
```

Achtung Automatisch definierte Projektionen werden dann zu partiellen Funktionen, falls nicht jeder Konstruktor alle Felder enthält:

```
u1 = Dozent  { name="Steffen", titel="Dr" }
u2 = Student { name="Max",  matrikel=666  }

> titel u1
"Dr"
> titel u2
"*** Exception: No match in record selector titel"
```

Records und Summentypen

Schlecht

```
data Obst = Apfel { preis,gewicht :: Double }
          | Birne { preis,gewicht :: Double }
```

Schlechter Stil, wenn jeder Konstruktor alle Felder hat, da dies zu unnötig vielen Fallunterscheidungen mit dupliziertem Code führt.

Besser

```
data Frucht = Apfel | Birne
data Obst   = Obst { preis,gewicht :: Double
                   , frucht :: Frucht }
```

Dank verschachtelter Patterns (ggf. Guards) verliert man nichts:

```
istTeurerApfel :: Obst -> Bool
istTeurerApfel Obst {frucht=Apfel,preis=p} = p > 7
istTeurerApfel _ = False
```

DisambiguateRecordFields

Feldnamen sollten innerhalb eines Moduls eindeutig sein; ansonsten überdeckt die letzte Definition eine vorangegangene Definition. Spracherweiterung `DisambiguateRecordFields` erlaubt Verwendung identischer Feldnamen in Situationen, welche aufgrund der Typisierung eindeutig sind.

```
module M where
  data S = MkS { x :: Int, y :: Bool }

module Foo where
  import M

  data T = MkT { x :: Int }

  ok1 MkS{ x=n } = n+1          -- Unambiguous
  ok2 n = MkT{ x = n+1 }        -- Unambiguous

  bad1 k = k { x = 3 }          -- Ambiguous
  bad2 k = x k                  -- Ambiguous
```

Dies ist besonders dann nützlich, wenn verschiedene Module verwendet werden, welche die gleichen Feldnamen definieren.

DuplicateRecordFields

Spracherweiterung `DuplicateRecordFields` erlaubt identische Feldnamen für verschiedene Records innerhalb eines Moduls. Hierfür findet nur eine sehr eingeschränkte Typinferenz statt:

```
data Foo = Foo { x :: Int, y :: Bool }
data Bar = Bar { z :: Int, y :: Double }

f :: Bar -> Double
f x = y (x :: Bar)   -- okay

g :: Bar -> Double
g x = y x            -- not accepted
```

Hinweis Subtyping zwischen Record-Typen gibt es in Haskell grundsätzlich nicht. Auch wenn ein Typ strikt mehr Felder besitzt als ein anderer, so bleiben es voneinander verschiedene Typen.

NamedFieldPuns

Es ist üblich, lokale Variablen in Pattern-Matches identisch zum Feld des Matchings zu benennen:

```
data Bar = Bar {a,b,c,d,e :: Int}
foo :: Bar -> Int
foo Bar{a=a, b=b, e=e} = a+b+e
```

GHC Spracherweiterung `NamedFieldPuns` vermeidet die dabei auftretenden Wiederholungen:

```
{-# LANGUAGE NamedFieldPuns #-}
foo Bar{a, b, e} = a+b+e
```

Ist eine Variable im Scope, deren Namen identisch zu einem Feldnamen ist, darf man diese auch zur Konstruktion verwenden:

```
let c = 1 in Bar {c}
-- anstatt
let c = 1 in Bar {c = c}
```

RecordWildCards

Wem das immer noch zu viel Tipparbeit ist, schreibt nur noch `..`
dank der Spracherweiterung `RecordWildCards`:

```
{-# LANGUAGE RecordWildCards #-}
data Bar = Bar {a,b,c,d,e :: Int}

foo :: Bar -> Int
foo Bar{ a = 1, .. } = b + c + d + e

bar :: Bar
bar = let { a=1; b=2; c=3; d=4; e=5 } in Bar {..}
```

Anstatt dem längerem äquivalenten Code:

```
foo Bar{ a=1, b=b, c=c, d=d, e=e } = b + c + d + e

bar = let   { a=1; b=2; c=3; d=4; e=5 }
      in Bar{ a=a, b=b, c=c, d=d, e=e }
```

RecordWildCards – Besonderheiten

Fallstrick: Fehlt eine Variable mit passenden Feldnamen im Sichtbarkeitsbereich, dann bleibt dieses Feld einfach undefiniert!

```
data Bar = Bar {a,b,c,d,e :: Int}

foo1 a c e = Bar {..}
foo2 a c e = Bar {a=a, c=c, e=e}
foo3 a c e = Bar {a=a, b=undefined, c=c, d=undefined,e=e}
```

Alle drei Definition sind für GHC äquivalent; aber für einen Menschen vielleicht nicht! $\Rightarrow$ Lesbarkeit kann dadurch leiden!

```
> c $ foo1 0 1 2
c
> b $ foo1 0 1 2
*** Exception: RecordWildCards.hs:12:13-20: Missing field
```

Record-Update mit WildCards ist dagegen gar nicht erlaubt.

Zusammenfassung Record-Syntax

Record-Syntax...

- ...erlaubt die Benennung von Konstruktor-Argumenten
- ...erlaubt, momentan unbenötigte Konstruktor-Argumente auszublenden, anstatt immer alle aufzuzählen zu müssen
- ...definiert automatisch (partielle) Projektionen
- ...bietet funktionale Updates durch Kopieren an
- ...kann die Wartbarkeit von Code erhöhen

Records sind in Haskell keine besondere Datenstruktur; es ist lediglich eine notationelle Vereinfachung!

Dementsprechend kennt Haskell auch kein Record-Subtyping

Metaprogrammierung

Manchmal möchte man den Quelltext eines Programms nicht direkt selbst programmieren, sondern durch ein Programm bearbeiten lassen. Weit verbreitet ist dazu z.B. der C-Präprozessor:

```
#define VERSION 2

 ...
#ifdef VERSION >= 3
  print "NEUESTE VERSION"
#else
  print "ALTE VERSION"
#endif
```

. . . auch für ghc mit Option -cpp

Template Haskell geht darüber hinaus:
Wir verwenden Haskell, um Haskell Code zu erstellen!

Template Haskell

Meta-Programmierung mit Spracherweiterung `TemplateHaskell` wird während des Kompiliervorgangs ausgeführt.
Zum Umschalten zwischen den Ebenen verwendet man:

Spleißen mit Dollar-ohne-folgendem-Leerzeichen `$( )`

Ein Datenobjekt, welches Code repräsentiert, wird damit während der Kompilierung wieder in Code umgewandelt und eingefügt.

– Klammer nur notwendig bei Argumenten

– Top-Level Splices dürfen $ weglassen

Quasi-Quoting mit Oxford-Klammern `[| |]`

Damit man nicht jeden Code-Schnipsel umständlich als Datenobjekt eingeben muss, kann man mit der Spracherweiterung `QuasiQuotes` Code innerhalb der Oxford-Klammern zu einem Datenobjekt umwandeln.

– Vereinfachte Quotes für existierende Namen:
`'funktion` `'Konstruktor` `''Typ`

Diese Wechsel der Ebenen dürfen auch verschachtelt werden!

Abstrakter Syntaxbaum für Haskell

Modul `Language.Haskell.TH` definiert zahlreiche Typen zur Modellierung des abstrakten Syntaxbaumes eines Haskell-Programms:

```
data Decl = FunD Name [Clause] | ValD Pat Body [Dec]
      | DataD Cxt Name [TyVarBndr] (Maybe Kind) [Con] [DerivClau
data Clause = Clause [Pat] Body [Dec]
data Body = NormalB Exp | GuardedB [(Guard,Exp)]

data Exp = VarE Name | LitE Lit | ConE Name | ParensE Exp
      | LamE [Pat] Exp | AppE Exp Exp | CasE Exp [Match] | ...

data Pat = VarP Name | LitP Lit | ConP Name [Pat] | ParensP Pat
      | WildP | TupP [Pat] | List [Pat] | AsP Name Pat | ...

data Lit = IntegerL Integer | CharL Char | StringL String | ...

data Type = VarT Name | ConT Name | ArrowT | TupleT Int | ...
```

Oxford-Klammer und Q-Monade

Es gibt mehrere Versionen der Oxford-Klammer:

1. `[e| ... |]` für Ausdrücke (expressions); liefert Typ `Q Exp`
2. `[d| ... |]` für Deklarationen; liefert Typ `Q [Decl]`
3. `[t| ... |]` für Typen; liefert Typ `Q Type`
4. `[p| ... |]` für Patterns; liefert Typ `Q Pat`

Weitere Tags für die Oxford-Klammer zum Quasi-Quoten von anderen Sprachen, wie z.B. HTML, CSS, JavaScript, CSS werden wir mit Webserver Yesod kennenlernen.

Code für Template Haskell wird in Monade `Q` erzeugt. Diese kümmert sich um alle Seiteneffekte der Code-Generierung, z.B. Typ-Kontext, Generierung frischer Namen, ...

Verwendet werden aber meist die vordeklarierten Typ-Synonyme `ExpQ`, `DecsQ`, `TypeQ` und `PatQ == Q Pat`

Template Haskell verwenden

Zur Verwendung von Template Haskell ist die Option

```
-XTemplateHaskell
```

oder besser das Pragma

```
{-# LANGUAGE TemplateHaskell #-}
```

notwendig.

Ein wichtiges Modul ist `Language.Haskell.TH`, welches u.a. bereitstellt:

```
runQ   :: Quasi m => Q a -> m a
runIO  :: IO a -> Q a
mkName :: String -> Name
reify  :: Name -> Q Info
```

Template Haskell verwenden

Die Seiteneffekte der Code-Generierung, z.B. frische Bezeichner, werden durch die `Q` Monade erfasst, welche durch Modul `Language.Haskell.TH` bereitgestellt wird.

```
runQ   :: Quasi m => Q a -> m a
mkName :: String -> Name
reify  :: Name -> Q Info
```

Beispiel

```
ghci -XTemplateHaskell -XQuasiQuotes
```

```
> let foo = [| \x -> 2+x |]
> runQ foo
LamE [VarP x_2] (InfixE (Just (LitE (IntegerL 2)))
                        (VarE GHC.Num.+) (Just (VarE x_2)))
> $(foo) 1
3
```

Template Haskell Beispiele

```
> runQ [| "Hi!" |]
LitE (StringL "Hi!")

> runQ [| 2 + 7 |]
InfixE (Just (LitE (IntegerL 2))) (VarE GHC.Num.+)
                                  (Just (LitE (IntegerL 7)))

> let f = [| \x ->  1 + x |]
> runQ f
LamE [VarP x_0] (InfixE (Just (LitE (IntegerL 1)))
                        (VarE GHC.Num.+) (Just (VarE x_0)))

> $(f) 6
7
> $( [| $(f) $ $(f) $ $( [| \z -> z*2 |]) 3 |] )
8
```

Beispiel: konstante *n*-stellige Funktion

Sinn & Zweck ist natürlich, die abstrakte Syntax mit Haskell zu manipulieren um viele Nützliche Deklarationen automatisch zu erstellen.

Beispiel: konstante *n*-stellige Funktion

```
> let cnst n s =
    return (LamE (replicate n WildP) (LitE (StringL s)))

> :t $(cnst 5 "x")
$(cnst 5 "x") :: t1 -> t2 -> t3 -> t4 -> t5 -> [Char]

> $(cnst 3 4.2) 1.1 2.2 3.3
4.2
```

Meist ist es jedoch mit QuasiQuotes etwas einfacher.

Beispiel: *i*-te Projektion *n*-Tupel

Generische Projektion des *i*-ten Elements eines *n*-Tuples:

```
projNI :: Int -> Int -> ExpQ
projNI n i = lamE [pat] rhs
  where pat = tupP (map varP xs)
        rhs = varE (xs !! (i - 1))
        xs  = [ mkName $ "x" ++ show j | j <- [1..n] ]
```

Für $i \leq n$ gilt `$(projNI n i) :: (t1,...,tn) -> ti`

Definition in *anderer* Datei darf dies dann Verwenden:

```
{-# LANGUAGE TemplateHaskell #-}
import MyLibraryContainingProjNI

main = print ( $(projNI 5 3) ('a','b','c','d','e') )
```

Beispiel: generisches Show (Anwendung)

```
{-# LANGUAGE TemplateHaskell #-}
import MyCustomShow
import Language.Haskell.TH

data MyData = MyData { foo :: String, bar :: Int }
$(listFields $ mkName "MyData")

main = print $ MyData { foo= "bar", bar= 5 }
```

```
> ./DemoCustomShow
foo="bar", bar=5
```

- Top-Level Deklarationen:
  - müssen nicht explizit gespleißt werden
  - haben nur Zugriff auf vorangegangene Deklarationen
- Ausführung von Funktion während der Kompilierung erfordert separate Modul-Datei für diese $\Rightarrow$ stage restriction error

Beispiel: generisches Show (Anwendung)

```
{-# LANGUAGE TemplateHaskell #-}
import MyCustomShow
import Language.Haskell.TH

data MyData = MyData { foo :: String, bar :: Int }
listFields ''MyData

main = print $ MyData { foo= "bar", bar= 5 }
```

```
> ./DemoCustomShow
foo="bar", bar=5
```

- Top-Level Deklarationen:
  - müssen nicht explizit gespleißt werden
  - haben nur Zugriff auf vorangegangene Deklarationen
- Ausführung von Funktion während der Kompilierung erfordert separate Modul-Datei für diese $\Rightarrow$ stage restriction error

Beispiel: generisches Show (Definition)

```
{-# LANGUAGE TemplateHaskell #-}
module MyCustomShow where
  import Data.List (intercalate)
  import Language.Haskell.TH

  listFields :: Name -> Q [Dec]
  listFields name = do
    TyConI (DataD _c _n _t _k [RecC _ fields] _d) <- reify name
    let fnames = map (\(fname,_b,_y) -> fname) fields
    let showField :: Name -> Q Exp
        showField name =
            [e|\x->s ++ "=" ++ show( $(varE name) x) |]
          where   s = nameBase name
    let showFields :: Q Exp
        showFields = listE $ map showField fnames
    [d|instance Show $(conT name) where
        show x = intercalate ", " (map ($ x) $showFields) |]
```

Beispiel behandelt nur Record-Constructors `RecC` im Pattern-Match!

Zusammenfassung Template Haskell

Template Haskell kann dazu verwendet werden:

- Typ-sicher stark generischen Code schreiben, z.B. Funktion `proj :: Int -> Int -> ExpQ` zur Projektion aus beliebigen n-Tupeln `$(projNI 3 2) :: (a, b, c) -> b`
- Implementation von generischen Mechanismen wie `deriving Show`
- Zur Manipulationen von anderen Programmiersprachen mit Haskell, insbesondere Domain-Specific-Languages (DSLs)
  z.B. Yesod-Webserver: HTML, CSS, JavaScript und JSON
- Erzeugte Splices kann man ansehen mit `ghc`-Optionen `-ddump-splices` oder `-dth-dec-file`
- Template Haskell seit 2002 von Tim Sheard und Simon Peyton Jones; QuasiQuotes seit 2007 von Geoffrey Mainland.

Funktionale Referenzen

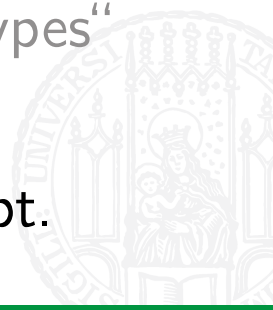
Bei der Programmierung entsteht öfters das Problem, einen kleinen Teil einer großen Datenstruktur abzuändern.

Wenn wir keine echten Referenzen innerhalb einer Monade verwenden wollen wie z.B. `STRef`, bleibt in einer rein funktionalen Sprache nur das Kopieren der gesamten Datenstruktur von der Wurzel bis zum geänderten Element; unveränderte Teile können übernommen werden, da Aliasing für uns ja unproblematisch ist.

Im jetzigen Vorlesungsabschnitt geht es dabei weniger um die Effizienz einer solchen Änderung, sondern viel mehr darum, wie man solche Änderungen einfach und elegant ausdrücken kann: mit einer **funktionalen Referenz** auf das zu ändernde Element.

a.k.a. „jQuery for abstract data types“

Linsen für abstrakte Datentypen machen auch in imperativen Sprachen Sinn, so gibt es z.B. Linsen-Bibliotheken für Javascript.



Problembeispiel

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
data Faction = Faction{ faction :: String, members :: [Person]}
cersei = Person "Cersei" $ Wealth 222 22
tyrion = Person "Tyrion" tw1
tw1    = Wealth 100 1
lannisters = Faction "Lannisters" [cersei,tyrion]
```

Problem: Tyrion gibt 7 Goldstücke aus

Warnung: Dies ist ein künstliches Beispiel!
Bei häufigen Änderungen wäre vielleicht andere Modellierung besser:

```
data WealthGold     = Data.Map.Map Name Gold
data WealthDiamonds = Data.Map.Map Name Diamond
data Person         = Person { name :: Name }
newtype Name        = Name String
newtype Gold        = Gold Int
newtype Diamomd     = Diamond Int
```

Problembeispiel

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
data Faction = Faction{ faction :: String, members :: [Person]}
cersei = Person "Cersei" $ Wealth 222 22
tyrion = Person "Tyrion" tw1
tw1    = Wealth 100 1
lannisters = Faction "Lannisters" [cersei,tyrion]
```

Problem: Tyrion gibt 7 Goldstücke aus

```
payGold  :: Int -> Person -> Person
payGold  m p = let w  = wealth p
                   w' = w { gold=(gold w) - m}
               in  p { wealth=w' }
```

```
> payGold  7 tyrion
Tyrion(93g,1d)
```

Funktioniert, aber sehr umständlich mit Record-Update-Syntax!

Problembeispiel

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
data Faction = Faction{ faction :: String, members :: [Person]}
cersei = Person "Cersei" $ Wealth 222 22
tyrion = Person "Tyrion" tw1
tw1    = Wealth 100 1
lannisters = Faction "Lannisters" [cersei,tyrion]
```

Problem: Tyrion gibt 7 Goldstücke aus

```
payGold' :: Int -> Person -> Person
payGold' m (Person n (Wealth g d)) =
  let w = (Wealth (g-m) d)
  in Person n w
```

```
> payGold  7 tyrion
Tyrion(93g,1d)
```

Ohne Record-Syntax sieht es einfach aus, bei vielen Feldern aber sehr umständlich!

Problembeispiel

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
data Faction = Faction{ faction :: String, members :: [Person]}
cersei = Person "Cersei" $ Wealth 222 22
tyrion = Person "Tyrion" tw1
tw1    = Wealth 100 1
lannisters = Faction "Lannisters" [cersei,tyrion]
```

Problem: Tyrion gibt 7 Goldstücke aus

Bequemer geht es mit **funktionalen Referenzen**, auch **Linsen** genannt:

```
> over (wealth.gold) (subtract 7) tyrion
Tyrion(93g,1d)

> over (members.(ix 1).wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(222g,22d),Tyrion(93g,1d)]

> over (members.traverse.wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(215g,22d),Tyrion(93g,1d)]
```

lens Package

Es gab und gibt verschiedene Versuche, Linsen zu implementieren. Ein Durchbruch gelang Edward Kmett MIRI Berkeley (US) 2012 mit dem Package `lens`.

Damit der gezeigte Code funktioniert, ist folgendes notwendig:

- `import Control.Lens`
- Feldern ein Underscore voranstellen – Konvention, kein muss
- Linsen automatisch erzeugen mit Template Haskell:

```
makeLenses ''Wealth
makeLenses ''Person
makeLenses ''Faction
```

Dadurch werden für die drei Typen `Wealth`, `Person` und `Faction` viele verschiedene Linsen erstellt.

ansehen mit `-ddump-splices` oder `-dth-dec-file`

Hintergrund

Die Idee von Linsen als **Put/Get-Paar**, oder auch View/Update, stammt aus der Datenbankforschung Ende 70er/Anfang 80er.

2005 taucht der Begriff „Linse“ in einer Arbeit von Benjamin Pierce Univ. of Pennsylvania, US, et al. auf. als Grundlage wird unter anderem eine Arbeit von Pierce & Hofmann aus 1995.

2009 beschrieb Twan van Laarhoven, Prof. Open Univ. Netherlands, in seinem Blog einen Typ, welcher jetzt `Lens'` genannt wird.

2010 erkannte Russell O'Connor die Verbindung zwischen seinen Multi-Linsen und den VL-Linsen, was zu Traversals führte.

2012 formulierte Edward Kmett MIRI Berkeley, US die benötigten Gesetze und schrieb die inzwischen populärste Linsen Bibliothek für Haskell Es gibt viel andere Bibliotheken mit anderen Ansätzen!

Grundidee

Eine Linse ist primär ein **Set/Get-Paar** bzw. View/Update-Paar:

```
> view wealth tyrion
(100g,1d)

> set wealth (Wealth 0 0) tyrion
Tyrion(0g,0d)
```

Grundidee

Eine Linse ist primär ein **Set/Get-Paar** bzw. View/Update-Paar:

```
> tyrion & view wealth
(100g,1d)

> tyrion & set wealth (Wealth 0 0)
Tyrion(0g,0d)
```

Hier wird auch gerne eine Schreibweise genommen,
bei der das Funktionsargument vorne steht!

Bei der Infix-1-Funktion `(&)` handelt es sich um `flip ($)`:

```
(&) :: a -> (a -> b) -> b      -- Modul Data.Function
x & f = f x

($) :: (a -> b) -> a -> b
f $ x = f x
```

Linsen als Paare

Eine Modellierungen von Linsen durch getter/setter-Paare ist prinzipiell möglich, z.B.

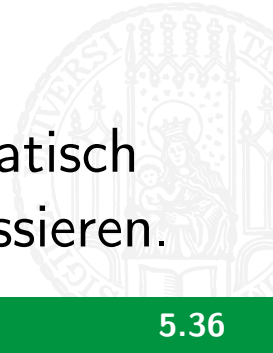
```
data LensPair s a = LensPair { view :: s -> a
                             , set  :: a -> s -> s }

over :: LensPair s a -> (a -> a) -> s -> s
over lens f s = set lens (f $ view lens s) s

lensPersonName   = LensPair name   (\x s-> s{name=x})
lensPersonWealth = LensPair wealth (\x s-> s{wealth=x})
```

```
> set lensPersonWealth (Wealth 9 1) tyrion
Tyrion(9g,1d)
```

Diese Modellierung von Linsen erweist sich aber etwas problematisch bezüglich Komposition, z.B. um das Gold einer Person zu fokussieren.



van Laarhoven Linsen

Twan van Laarhoven schlug 2009 folgende Definition vor:

```
{-# LANGUAGE Rank2Types #-}

type Lens' s a =
  forall f. Functor f => (a -> f a) -> (s -> f s)

view :: Lens' s a -> s -> a
view l = getConst . l Const

set :: Lens' s a -> a -> s -> s
set l a = over l (const a)

over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

Rank-N types

Spracherweiterung `RankNTypes` erlaubt Typen, bei denen ∀-Quantoren für Typvariablen mitten im Typ auftauchen, und nicht nur wie bisher ganz vorne.

Pathologisches Beispiel

```
-- foo :: (forall a . a) -> b
foo :: (forall a . a -> a) -> (b -> b)
foo x = x x

> (foo (foo id)) 42
  42
```

Keine der beiden Typsignaturen kann von GHC inferiert werden.

`foo` hat keinen prinzipalen (allgemeinsten) Typ.
Die Typinferenz wird unentscheidbar!
⇒ Typsignaturen werden notwendig.

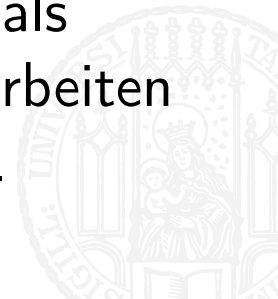
Rank-N Types

Ein ∀-Quantor auf der rechten Seite eines Pfeiles ist harmlos: `forall a. a-> (forall b. b->a)` ist äquivalent zu `forall a b. a->b->a`. Beides sind Rang-1 Typen, deren Bedeutung identisch ist zu dem bisherigen Typ `a->b->a`

Ein ∀-Quantor auf der linken Seite eines Pfeiles kann nicht verschoben werden und erhöht den Rank des Typen:

```
forall a. (forall b.b->b) -> a -> a
```

Bedeutung: Hier wird eine Funktion als Argument gefordert, welche mit jeden beliebigen Typen umgehen kann! muss `id` sein?!
Im Gegensatz dazu fordert `forall a b. (b->b) -> a -> a` als Argument nur eine Funktion, welche irgendein spezielles `b` verarbeiten kann. Für jedes `b` darf eine andere Funktion übergeben werden.



Beispiele Rank-N Types

Beispiele, wobei die Nummer im Namen dem Rang entspricht:

```
f1 :: forall a b. a -> b -> a
g1 :: forall a b. (Ord a, Eq  b) => a -> b -> a

f2 :: (forall a. a->a) -> Int -> Int
g2 :: (forall a. Eq a => [a] -> a -> Bool) -> Int -> Int

f3 :: ((forall a. a->a) -> Int) -> Bool -> Bool
```

Wie bei der Ordnung eines Typs bezieht sich der Rang auf die maximale Tiefe, in der ein $\forall$-Quantor links innerhalb eines Pfeil-Typen auftritt.

Die „leere“ Identity Monade A5-3 & Wdh. 4.61

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f (Identity x) = Identity $ f x

instance Applicative Identity where
  pure :: a -> Identity a
  pure = Identity
  (<*>) :: Identity (a->b) -> Identity a -> Identity b
  Identity f <*> Identity x = Identity $ f x

instance Monad Identity where
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  Identity x >>= mf = mf x
```

`(>>=)` entspricht `flip ($)`, vom newtype-Konstruktor abgesehen.

Die „leere“ Identity Monade A5-3 & Wdh. 4.61

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f mx = Identity $ f $ runIdentity mx

instance Applicative Identity where
  pure :: a -> Identity a
  pure = Identity
  (<*>) :: Identity (a->b) -> Identity a -> Identity b
  mf <*> mx = Identity $ (runIdentity mf)(runIdentity mx)

instance Monad Identity where
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  mx >>= mf = mf $ runIdentity mx
```

`(>>=)` entspricht `flip ($)`, vom newtype-Konstruktor abgesehen.

Die „leere“ Identity Monade A5-3 & Wdh. 4.61

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f mx = Identity $ f $ runIdentity mx

instance Applicative Identity where
  pure :: a -> Identity a
  pure = Identity
  (<*>) :: Identity (a->b) -> Identity a -> Identity b
  mf <*> mx = Identity $ (runIdentity mf)(runIdentity mx)

instance Monad Identity where
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  mx >>= mf = mf $ runIdentity mx
```

(<$>) entspricht ($), vom newtype-Konstruktor abgesehen.

Der „konstante“ Const Functor


```
newtype Const a b = Const {getConst :: a}

instance Functor (Const m) where
  fmap :: (a -> b) -> Const m a -> Const m b
  fmap _ (Const x) = Const x
instance Monoid m => Applicative (Const m) where
  pure :: a -> Const m a
  pure = const $ Const mempty
  (<*>) :: Const m (a->b) -> Const m a -> Const m b
  Const x <*> Const y = Const $ x `mappend` y
```

`Const a b` ist ein applikativer Funktor, dem man sich wie eine Container über `b` vorstellen kann, welcher kein `b` enthält, dafür als Kontext einen Wert des Typs `a` hat.

`b` bezeichnet man auch als **Phantom-Typen**, da Werte von `Const a b` nie `b`-Werte enthalten.

```
  traverse _ (Const m) = pure $ Const m
```

Der „konstante“ Const Functor

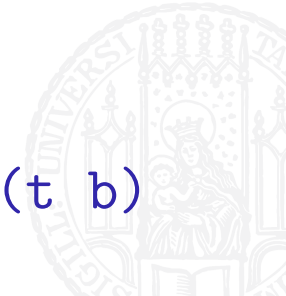

```
newtype Const a b = Const {getConst :: a}
```

`Const a b` ist keine Monade, aber nützlich als Instanz der Klassen `Foldable` und `Traversable`.

In Übungsaufgabe A5-3 haben wir gesehen wie dank `Const` aus jeder `Traversable`-Instanz ein `Foldable`-Instanz ableitbar ist:

```
foldMapDefault::(Traversable t,Monoid c)=>(a->c)->t a->c
foldMapDefault f = getConst . traverse (Const . f)
```

```
instance Foldable (Const m) where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap _ _ = mempty
instance Traversable (Const m) where
  traverse :: Applicative f=> (a->f b)-> t a-> f(t b)
  traverse _ (Const m) = pure $ Const m
```

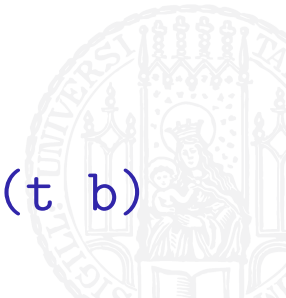


Der „konstante“ Const Functor


```
newtype Const a b = Const {getConst :: a}

instance Functor (Const m) where
  fmap :: (a -> b) -> Const m a -> Const m b
  fmap _ (Const x) = Const x
instance Monoid m => Applicative (Const m) where
  pure :: a -> Const m a
  pure = const $ Const mempty
  (<*>) :: Const m (a->b) -> Const m a -> Const m b
  Const x <*> Const y = Const $ x `mappend` y

instance Foldable (Const m) where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap _ _ = mempty
instance Traversable (Const m) where
  traverse :: Applicative f=> (a->f b)-> t a-> f(t b)
  traverse _ (Const m) = pure $ Const m
```



van Laarhoven Linsen

Twan van Laarhoven schlug 2009 folgende Definition vor:

```
{-# LANGUAGE Rank2Types #-}

type Lens' s a =
  forall f. Functor f => (a -> f a) -> (s -> f s)

view :: Lens' s a -> s -> a
view l = getConst . l Const

set :: Lens' s a -> a -> s -> s
set l a = over l (const a)

over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

Defintion eigener Linsen

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
```

Die van-Laarhoven Linsen für unsere Datentypen sind:

```
-- forall f. Functor f=> (Int->f Int)-> Wealth-> f Wealth
lWgold :: Lens' Wealth Int
lWgold     k (Wealth g d) = (\g' -> Wealth g' d) <$> k g

lWdiamonds :: Lens' Wealth Int
lWdiamonds k (Wealth g d) = (\d' -> Wealth g d') <$> k d

lPname :: Lens' Person String
lPname     k (Person n w) = (\n' -> Person n' w) <$> k n

lPwealth :: Lens' Person Wealth
lPwealth   k (Person n w) = (\w' -> Person n w') <$> k w
-- forall f.Functor f=> (Wealth-> f Wealth)-> Person-> f Person
```

Immer gleiches Muster $\Rightarrow$ automatisch mit TemplateHaskell erzeugen

Konvention des lens Paketes

Paket `lens` stellt TemplateHaskell zur Verfügung, um diese Linsen automatisch erzeugen: Eine Linse für jedes Record-Feld, welches mit Unterstrich beginnt. Die Linse heißt wie das Feld ohne Unterstrich:

```
data Wealth  = Wealth { _gold, _diamonds :: Int }
data Person  = Person { _name :: String, _wealth :: Wealth }
data Faction = Faction{ _faction :: String, _members :: [Person]}
makeLenses ''Wealth
$(makeLenses (mkName "Person"))
makeLenses ''Faction
```

```
> :type gold
gold :: Functor f => (Int -> f Int) -> Wealth -> f Wealth
> :type wealth.gold
wealth.gold :: Functor f => (Int -> f Int) -> Person -> f Person
```

Dies ist die Standard-Option; eigene Benennungen wählbar mit

```
makeLensesWith :: LensRules -> Name -> DecsQ
```

view mit Const

```
Const    :: a -> Const a b
getConst :: Const a b -> a
view::(forall f.Functor f=>((a->f a)->(s->f s))) -> s -> a
view l = getConst . l Const

data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1     -- für ein tw1::Wealth

lPname :: Functor f=> (String-> f String)-> Person-> f Person
lPname k (Person n w) = (\n' -> Person n' w) <$> k n
```

```
> view lPname tyrion
> getConst . lPname Const $ tyrion
> getConst $ lPname Const (Person "Tyrion" tw1)
> getConst $ (\n'-> Person n' tw1) <$> Const "Tyrion"
> getConst $ (\n'-> Person n' tw1) <$> Const "Tyrion"
> getConst $ Const "Tyrion"
"Tyrion"
```

over mit Identity

```
Identity    :: a -> Identity a
runIdentity :: Identity a -> a
over::(forall f.Functor f=>((a->f a)->(s->f s)))->(a->a)->s->s
over l m = runIdentity . l (Identity . m)

data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1     -- für ein tw1::Wealth

lPname :: Functor f=> (String-> f String)-> Person-> f Person
lPname k (Person n w) = (\n' -> Person n' w) <$> k n
```

```
> over lPname reverse tyrion
> (runIdentity . lPname (Identity.m)) tyrion
> runIdentity $ lPname (Identity . reverse) (Person "Tyrion" tw1)
> runIdentity $ (\n'-> Person n' tw1) <$> (Identity $ reverse "Tyrion")
> runIdentity $ (\n'-> Person n' tw1) <$> (Identity "noiryT")
> runIdentity $ Identity $ (\n' -> Person n' tw1) "noiryT"
> runIdentity $ Identity $ Person "noiryT" tw1)
noiryT(100g,1d)
```

van Laarhoven Linsen Komposition

Komposition von van-Laarhoven-Linsen entspricht einfach umgedrehter Funktionskomposition:

```
type Lens' s a =
  forall f. Functor f => (a -> f a) -> (s -> f s)

compose :: Lens' b c -> Lens' a b -> Lens' a c
compose r s = s . r

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

Einsetzen des Typsynonyms ergibt einfach:

```
compose :: forall a b c f. Functor f
        => ((c -> f c) -> (b -> f b))
        -> ((b -> f b) -> (a -> f a))
        -> ((c -> f c) -> (a -> f a))
```

van Laarhoven Linsen Komposition

Man Linsen mit Funktionskomposition zusammensetzen:

```
> tyrion & view (lPwealth.lWgold)
100
> :type (lPwealth.lWgold)
(lPwealth.lWgold) :: Functor f =>
  (Int -> f Int) -> Person -> f Person
```

Allerdings mit verdrehter Reihenfolge: In der funktionalen Welt ist die äußere Funktion normalerweise auf der rechten Seite des Punktes:

```
> negate.length $ [1..3]
-3
```

Die Komposition von Linsen erinnert dagegen an die Accessor-Verkettung in objektorientierten Sprachen wie etwa Java:

```
> tyrion^.wealth.gold
100
```

wobei hier `(^.)` ein Infix-Synonym für `view` ist.

Infix-Operatoren

Das `lens` Paket definiert mehr als 100 Infix Operatoren:

```
view == (^.) :: s -> Lens' s a -> a
set  == (.~) :: Lens' s a -> a -> s -> s
over == (%~) :: Lens' s a -> (a -> a) -> s -> s
```

Die Benennung hält sich an folgende Konventionen:

- `^` — kennzeichnet Getter-ähnliche Operatoren
- `~` — Setter-ähnliche
- `.` — grundlegende Operatoren
- `%` — Operatoren mit Funktionen als Parameter
- `=` — Setter-Variante für `State` Monade

Viele Operatoren für kleine Bequemlichkeiten:

```
(&&~) :: ASetter' s Bool -> Bool -> s -> s
l &&~ n = over l (&& n)
(<>~) :: Monoid a => ASetter' s a -> a -> s -> s
l <>~ m = over l (`mappend` m)
```

Polymorphe Linsen

Bis jetzt haben wir folgende Typen:

```
type AGetter' s a = forall r.(a -> Const r  a)-> s-> Const r s
type ASetter' s a =          (a -> Identity b)-> s-> Identity t
type Lens'    s a = forall f. Functor f =>
                             (a -> f a)       -> s-> f t
```

Für polymorphe Datentypen wie z.B.:

```
data Person nty wty = Person { name::nty, wealth::wty }
```

Der Typ für die Felder `name` und `wealth` ist nun austauschbar.

Dazu ist folgende Verallgemeinerung notwendig

```
type Lens'    s   a   = Lens s s a a
type Lens     s t a b = forall f. Functor f =>
                        (a -> f        b) -> (s -> f        t)
type SSetter s t a b = (a -> Identity b) -> (s -> Identity t)
```

Ansonsten ändert sich nichts. O'Connor 2010

Linsen Gesetze

Pierce formulierte bereits Gesetze für „very well behaved lens“:

1. Man bekommt zurück, was man hineintut:
   ```
   view l (set l v s) == v
   ```
2. Das vorhandene neu zu setzen ändert nichts:
   ```
   set l (view l s) s == s
   ```
3. Nur das letzte Setzen zählt:
   ```
   set l v2 (set l v1 s) == set l v2 s
   ```

Kmett betrachtete die Konsequenzen aus diesen Gesetzen und formulierte Varianten für weitere Optiken, welche wir jetzt noch betrachten werden.

Traversal

Was passiert, wenn wir statt `Functor` sogar `Applicative` einfordern? Wir erhalten eine *schwächere* `Traversal`-Optik:

```
type Traversal s t a b = forall f. Applicative f =>
                                   (a -> f b) -> (s -> f t)
type Lens      s t a b = forall f. Functor     f =>
                                   (a -> f b) -> (s -> f t)
```

Achtung: Jede `Lens` ist auch ein `Traversal`, nicht umgekehrt!

Dies mag auf dem ersten Blick merkwürdig erscheinen; wenn man den Typ aber umgangssprachlich vorliest, wird es klar:

Traversal Ist eine Funktion, welche mit jedem beliebigen Typen der Klasse `Applicative` umgehen können muss.

Lens Ist eine Funktion, welche mit jedem beliebigen Typen der Klasse `Functor` umgehen können muss.

Wegen `Functor` $\supset$ `Applicative` muss `Lens` mehr Typen beherrschen!

Traversal

Während eine Linse immer genau ein Element fokussiert, hat eine `Traversal`-Optik immer mehrere Elemente im Focus.

Funktion `traverse` aus der Klasse `Traversable` hat bereits einen zu van-Laarhoven-Linsen kompatiblen Typ:

```
traverse :: (Traversable t, Applicative f) =>
                          (a -> f b) -> t a -> f (t b)
```

Anwendungsbeispiel:

```
data Faction = Faction{ faction::String, members::[Person] }
```

```
> over' (lFmembers.traverse.lPwealth.lWgold) (*10) lannisters
Lannisters[Cersei(2220g,22d),Tyrion(1000g,1d)]
```

Wir nutzen hier natürlich aus, dass Listen `Traversable` sind.

Traversal

Dieses Beispiel nutzt selbst-definierte van-Laarhoven-Linsen. Hier ergibt sich ein Problem, dass wir `over` einen zu allgemeinen Typ gegeben hatten. Es klappt aber mit:

```
over' :: Traversal s s a a -> (a -> a) -> s -> s
```

Das `lens`-Paket definiert zur Umgehung für solche Probleme zahlreiche Klassen und Typsynonyme.

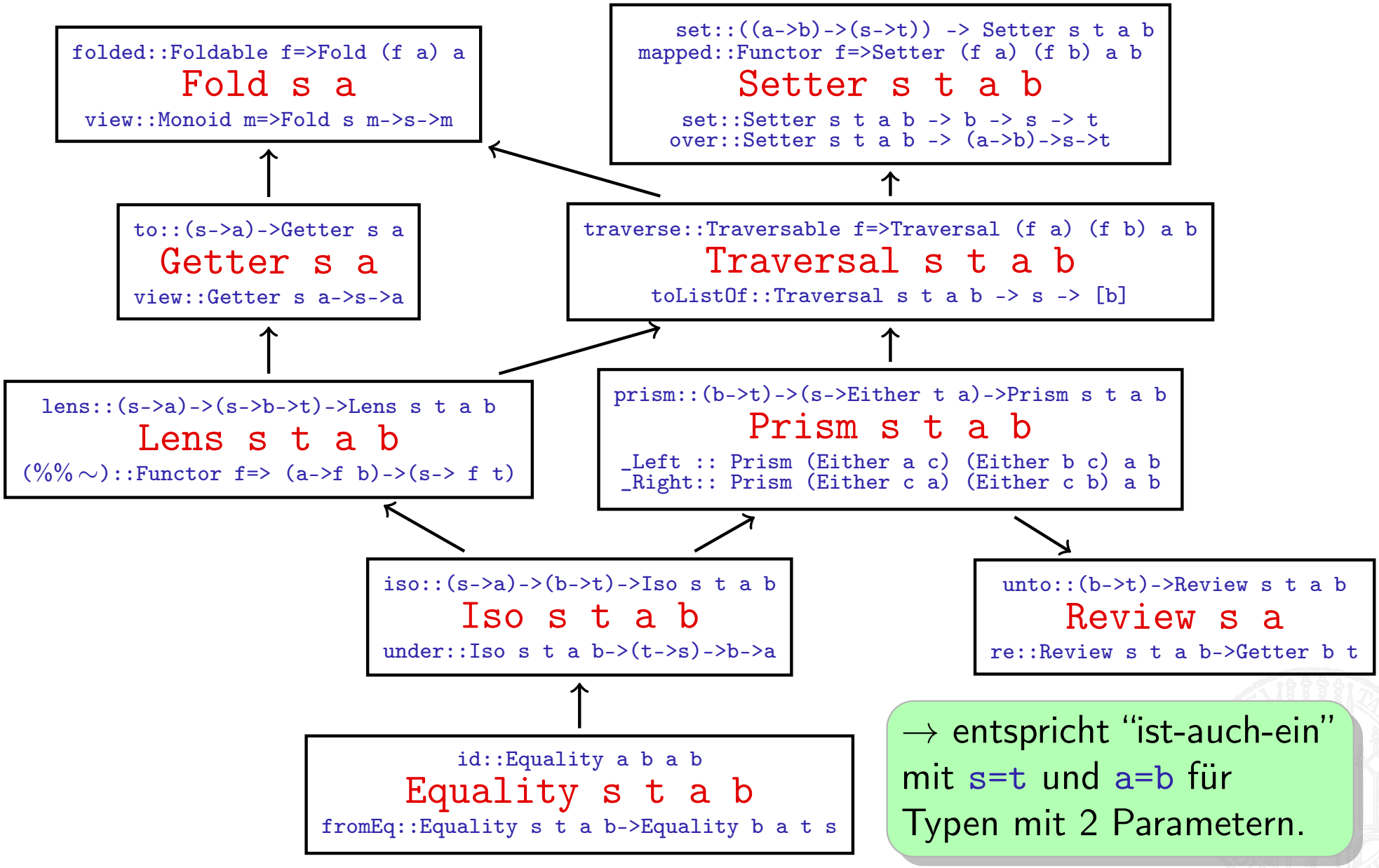
Anwendungsbeispiel:

```
data Faction = Faction{ faction::String, members::[Person] }
```

```
> over' (lFmembers.traverse.lPwealth.lWgold) (*10) lannisters
Lannisters[Cersei(2220g,22d),Tyrion(1000g,1d)]
```

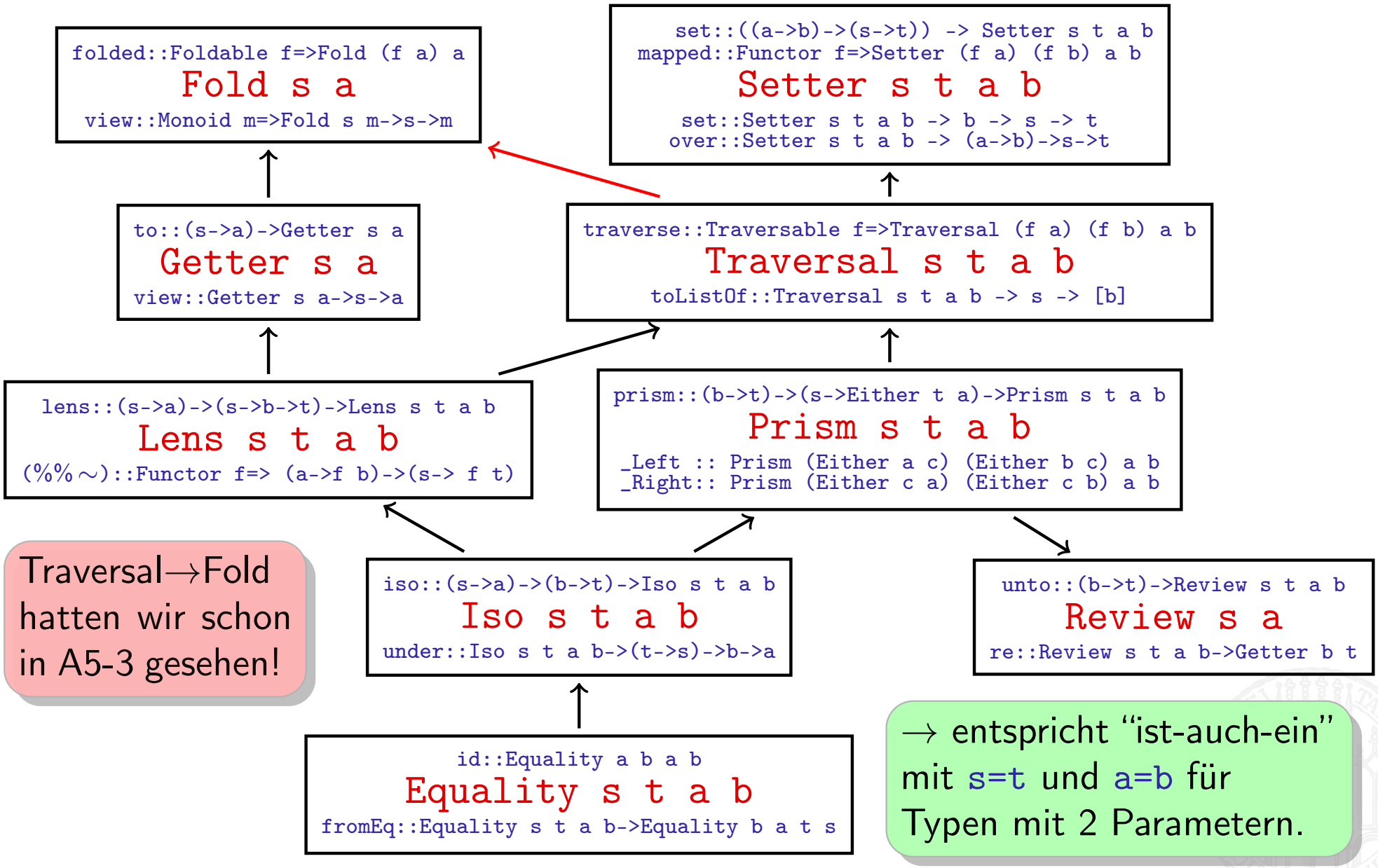
Wir nutzen hier natürlich aus, dass Listen `Traversable` sind.

Lens Hierarchy



→ entspricht “ist-auch-ein” mit s=t und a=b für Typen mit 2 Parametern.

Lens Hierarchy



Prismen

Linsen fokussieren immer genau ein inneres Element.
Prismen fokussieren dagegen ein oder kein Element.

```
type Prism s t a b= forall p f. (Choice p, Applicative f)=>
  p a (f b) -> p s (f t)
```

Linsen arbeiten mit Produkt-Typen wir Paaren oder Records;
Prismen kümmern sich um Summen-Typen wie `Maybe` oder
`Either`. Ein Prisma kann also Pattern-Matching kodieren:

```
> set _Just 7 (Just "Sieben")
Just 7
> set _Just 7 Nothing
Nothing
> over _Right (3*) (Right 7)
Right 21
> over _Right (3*) (Left 7)
Left 7
```

Prismen

Prismen sind aber keine Getter, da ja möglicherweise kein Fokus existiert. Stattdessen erhalten wir ein `preview`:

```
> preview _Left (Left 42)
Just 42
> preview _Left (Right 42)
Nothing
```

Dafür lassen sich Prismen invertieren:

```
> review _Left 69
Left 69
```

Gesetze

1. Eine Vorschau auf eine Rückschau gelingt immer mit dem gleichem Ergebnis:
   `preview p (review p x) == Just x`
2. Eine Rückschau auf eine *erfolgreiche* Vorschau sollte ebenfalls gelingen: `review p <$> preview p z == Just z`

Iso

Die Optik `Iso` ist für Isomorphismen, z.B. um auszudrücken, dass die beiden Typen `Maybe a` und `Either () a` jederzeit ineinander überführt werden können.

Für einen Isomorphismus braucht man zwei Funktionen `fw :: s -> a` und `rw :: s -> a`, so dass gilt:

```
fw . rw == id
rw . fw == id
```

damit können wir ein `Iso`-Optik erzeugen:

```
iso :: (s -> a) -> (b -> t) -> Iso s t a b
```

Dies kann nützlich sein, wenn wir verschiedene Repräsentationen benötigen, z.B. eine Iso-Optik welche uns den beliebigen Wechsel zwischen Werten eines Haskell-Datentyp und dessen XML- und/oder JSON-Repräsentationen erlaubt.

Vorteile an Paket lens

Paket `lens` ist ein schweizer Taschenmesser, z.B. macht es unser TemplateHaskell `projNI` von Folie 5.27 überflüssig, denn es liefert Typ-sichere Linsen, welche in beliebige Tupel fokussieren:

Die Tupel müssen nur groß genug sein

```
> view _3 ('a','b','c')
'c'
> view _3 ('a','b','c','d','e','f','g')
'c'
> set _3 'z' ('a','b','c','d','e','f','g','h')
('a','b','z','d','e','f','g','h')
> :type _3
_3 ::(Field3 s t a b, Functor f)=> (a -> f b)-> s -> f t
```

Der Trick liegt hier in der Klasse `Field3`, eine Klasse für alle Tupel-artigen Container, welche ein drittes Element haben.

TemplateHaskell `makeClassy` generiert Linsen und solche Klassen.

Kritik am lens Paket

Die `lens` Bibliothek ist sehr komplex:

- Selbst einfache Beschreibungen wirken einschüchternd oder verwenden Kategorientheorie
- Verwirrung durch unüberschaubare Flut von Klassen, Typen, Funktionen und Operatoren
- Sehr komplexe Typen und zahlreiche Typsynonyme liefern hoffnungslose Typfehler
- Installation des Pakets dauert lange, aufgrund vieler Abhängigkeiten

$\Rightarrow$ Es gibt einige alternative Linsen-Bibliotheken, welche sich darauf konzentrieren, möglichst leichtgewichtig zu sein, z.B. `fclabels`

Nützliche vordefinierte Optiken

Linsen

```
(_1) :: Field1 s t a b => Lens s t a b -- Projektionen
(_2) :: Field2 s t a b => Lens s t a b -- Projektionen
```

Traversals

```
firstOf :: Traversal' s a -> Maybe a -- == preview
lastOf  :: Traversal' s a -> Maybe a
```

Prismen

```
_Nothing :: Prism' (Maybe a)          ()
_Just    :: Prism  (Maybe a) (Maybe b) a b
_Left    :: Prism (Either a c) (Either b c) a b
_Right   :: Prism (Either c a) (Either c b) a b
_Cons    :: Prism  [a] [b] (a,[a]) (b,[b])
```

Zusammenfassung

- „Lenses are ‘Co‘-state Comonad Coalgebras“

Zusammenfassung

- Funktionale Referenzen, auch bekannt als Linsen, ermöglichen Zugriff auf tiefer liegende Teile von abstrakten Datenstrukturen
- Linsen sind Werte, d.h. Referenzen in Datenstrukturen können übergeben, manipuliert und mit `(.)` zusammengesetzt werden

  $\Rightarrow$ First-Class Values
- `Lens s t a b`: Dabei ist `s` der äußere Datentyp `s`, und `a` der innere Datentyp.
  Man kann `a` zu `b` verändern. Dann ist `t` der Datentyp, den man erhält, wenn man in `s` den Typ `a` durch `b` ersetzt.
- Rank-N Types und Typ-Klassen ermöglichen etwas, dass sich sehr ähnlich wie Subtyping verhält

  z.B. jede Linse ist ein Getter, jedes Prisma ein Traversal,...
- Verschiedene Implementierungen von Linsen sind möglich/erhältlich, auch für andere Programmiersprachen