# Spracherweiterungen von Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik


Letzte Änderung der Folien: 4. Februar 2022 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

# Einleitung

Ziel des Kapitels

- Spracherweiterungen ggü. dem Haskell-Standard kennenlernen
- Insbesondere solche, die häufig verwendet werden
- Themen sind: Multiparameter-Klassen, Funktionale Abhängigkeiten, Typfamilien, GADTs, DataKinds, existentielle Typen, View-Patterns, Pattern-Synonyme, OverloadedStrings
- Wir geben nur einen Überblick und orientieren uns an Beispielen, aber behandeln die Erweiterungen nicht in aller Breite
- GHC hat viele Erweiterungen, mit

  ```
  ghc --supported-extensions
  ```

  kann man sich alle anzeigen lassen

# Multiparameter-Typklassen

- `MultiParamTypeClasses` erlaubt Typklassen mit mehreren Parametern
- Beispiel: Monaden mit Variablen (Speicherplätze):

```
class Monad m => VarMonad m v where
  newRef   :: a -> m (v a)
  readRef  :: v a -> m a
  writeRef :: v a -> a -> m ()
```

- Generische Funktion, die Speicherzelle um 1 erhöht:

```
addOne  :: (VarMonad m v, Num a) => v a -> m ()
addOne v = do x <- readRef v
              writeRef v $ x+1
```

# Multiparameter-Typklassen (2)

Instanzen für `VarMonad`:

```
instance VarMonad IO IORef where
    newRef   = newIORef
    readRef  = readIORef
    writeRef = writeIORef

instance VarMonad STM TVar where
    newRef       = newTVar
    readRef      = readTVar
    writeRef     = writeTVar

instance VarMonad IO MVar where
    newRef   = newMVar
    readRef  = readMVar
    writeRef m v = swapMVar m v >> return ()
```

# Multiparameter-Typklassen (3)

Das folgende Programm ist z.B. möglich:

```
main = do
  ioRef <- newIORef 3; mvar <- newMVar 5; stmRf <- newTVarIO 7
  addOne ioRef
  addOne mvar
  atomically $ addOne stmRf
  print =<< readRef ioRef     -- Ausgabe: "4"
  print =<< readRef mvar      -- Ausgabe: "6"
  print =<< readTVarIO stmRf  -- Ausgabe: "8"
```

# Typklassen vs. Multiparameter-Typklassen

**Sichtweise**:

- Typklasse = Menge von Typen
- Multiparameter-Typklasse = Relation auf Typen

# Typklassen vs. Multiparameter-Typklassen

**Sichtweise**:

- Typklasse = Menge von Typen
- Multiparameter-Typklasse = Relation auf Typen

**Probleme**

- Typ-Inferenz deutlicher komplizierter und teilweise unklar
- `VarMonad` hängt der zweite Parameter `v` von `m` ab (z.B. geht nicht: `IO` und `TVar`)
- Zusammenhang wird durch Multiparameter-Typklassen noch nicht erkennbar

# Funktionale Abhängigkeiten

- Erweiterung `FunctionalDependencies` erlaubt es bei Typklassendeklaration die Funktionale Abhängigkeit `m -> v` festzulegen.
- Besagt: Typ von `v` wird eindeutig durch den Typ von `m` bestimmt
- Das ergibt:

```
class Monad m => VarMonad m v | m -> v where
  newRef   :: a -> m (v a)
  readRef  :: v a -> m a
  writeRef :: v a -> a -> m ()
```

- Beachte: Nun kann man nicht mehr beide Instanzen

```
instance VarMonad IO  IORef   where ...
instance VarMonad IO  MVar    where ...
```

angeben

# Funktionale Abhängigkeiten (2)

Allgemeiner:

```
class A a b                    -- bel. binäre Relationen
class B a b | a -> b           -- partielle Funktionen
class C a b | a -> b, b -> a   -- 1-zu-1-Abbildungen
```

# Funktionale Abhängigkeiten (2)

Allgemeiner:

```
class A a b                    -- bel. binäre Relationen
class B a b | a -> b           -- partielle Funktionen
class C a b | a -> b, b -> a   -- 1-zu-1-Abbildungen
```

Beispiele

```
instance A Char Bool  -- erlaubt
instance A Char Int   -- erlaubt
instance A Int Bool   -- erlaubt

instance B Char Bool
instance B Char Int -- verboten wegen a -> b und Char -> Bool schon definiert
instance B Int Bool -- erlaubt

instance C Char Bool
instance C Char Int -- verboten wegen a -> b (und Char -> Bool schon definiert)
instance C Int Bool -- verboten wegen b -> a (und Bool -> Char schon definiert)
```

# Funktionale Abhängigkeiten (3)

Funktionale Abhängigkeiten können auch mehrere Parameter umfassen:

```
class D a b c d e f | a -> b c, d e -> f
```

- `a` bestimmt eindeutig `b` und `c`
- Die Kombination aus `d` und `e` bestimmt eindeutig `f`

# Funktionale Abhängigkeiten (3)

Funktionale Abhängigkeiten können auch mehrere Parameter umfassen:

```
class D a b c d e f | a -> b c, d e -> f
```

- `a` bestimmt eindeutig `b` und `c`
- Die Kombination aus `d` und `e` bestimmt eindeutig `f`

Beispiele:

```
instance D Bool Int Char Float Double Int
instance D Bool Int Char Float Double Integer     -- nicht erlaubt wegen d e -> f
instance D Bool Int Char Float Float Int          -- erlaubt
instance D Bool Integer Char Float Double Integer -- nicht erlaubt wegen a -> b c
instance D Bool Int Integer Float Double Integer  -- nicht erlaubt wegen a -> b c
```

# Typfamilien

- Typklassen überladen Funktionen
- Typfamilien überladen Datentypen
- Typfamilien sind über Erweiterung `TypeFamilies` verfügbar

Beachte (analog zu Funktionen und Klassen):

Implementierung des Datentyps darf je nach Instanz unterschiedlich sein.

Typfamilien sind sehr mächtige, ausdrucksstark, gestatten bereits Funktionen auf Typ-Ebene

# Typfamilien: Beispiele (1)

```
{-# LANGUAGE TypeFamilies #-}
class Mutation m where
  type Ref m  :: * -> *
  newRef      :: a -> m (Ref m a)
  readRef     :: Ref m a -> m a
  writeRef    :: Ref m a -> a -> m ()

instance Mutation STM where
  type Ref STM = TVar
  newRef       = newTVar
  readRef      = readTVar
  writeRef     = writeTVar

instance Mutation IO where
  type Ref IO  = IORef
  newRef       = newIORef
  readRef      = readIORef
  writeRef     = writeIORef
```

- `Ref` ist Funktion auf Typebene
- `Ref m` berechnet den Typ Referenz.
- Nur eine Instanz für `IO` möglich (entweder `IORef` oder `MVar`)

# Typfamilien: Beispiele (2)

```
{-# LANGUAGE TypeFamilies #-}
class Add a b where
  type SumTy a b
  add :: a -> b -> SumTy a b
```

- `SumTy` berechnet Typ der Summe aus Typen der Argumente
- Instanzen:

```
instance Add Integer Double where
  type SumTy Integer Double = Double
  add x y = fromIntegral x + y

instance (Num a) => Add a a where
  type SumTy a a = a
  add x y = x + y

instance (Add Integer a) => Add Integer [a] where
  type SumTy Integer [a] = [SumTy Integer a]
  add x y = map (add x) y
```

# Top-level Typfamilien

- Bisher: Mit Klassen assoziierte Typfamilien.
- Es gibt aber auch selbständige Typfamilien
- Beispiel:

```
type family G a where
  G Int = Bool
  G a   = Char
```

- `G` ist Funktion auf Typ-Ebene, die `Int` `Bool` und alle anderen Typen auf `Char` abbildet

# Offene Deklarationen

- Die Deklaration

```
type family G a where
  G Int = Bool
  G a   = Char
```

  ist eine geschlossene Deklaration (alle Fälle sind definiert, keine Änderung möglich)

- Es gibt auch offene (erweiterbare) Deklarationen:

```
type family F a b :: * -> *
type instance F Int Bool = Maybe
type instance F Int Int = Either ()
```

- `F` bildet `Int` und `Bool` auf `Maybe`und `Int` und `Int` auf `Either ()` ab

- Weitere Fälle können hinzugefügt werden (daher offen)

# Datentyp-Familien

```
type family F a b :: * -> *
type instance F Int Bool = Maybe
type instance F Int Int = Either ()
```

- Deklaration definiert Typsynonyme: (`F Int Bool` ist Synonym zu `Maybe` usw.).

  ```
  fun :: F Int Bool Bool
  fun = Just True
  ```

- Neben Typfamilien für Typsynonyme gibt es auch Typfamilien für Datentypen (passend zu `data` und `newtype`).
- Auch Datentypfamilen können assoziiert mit Typklassen oder auf Top-Level auftreten.

# Datentyp-Familien: Beispiel

Typfamilie für voll ausgewertete Listen:

- Für Elementtyp `Char` wie üblich
- Für Elememttyp `()` reicht es die Länge zu speichern

```
-- Deklaration der Datentyp-Familie

data family XList a

-- Instanz für Char:

data instance XList Char = XCons !Char !(XList Char) | XNil

-- Instanz für ():

data instance XList () = XListUnit !Int
```

# Datentyp-Familien: Beispiel (2)

Nicht möglich:

```
foo :: XList a -> Int -- ERROR
foo XNil          = 0
foo (XCons _ t)   = 1 + foo t
foo (XListUnit n) = n
```

Da `XList` offen ist, können neue Fälle nach Hinzufügen weiterer Instanzen auftreten können.

Möglich jedoch:

```
foo1 :: XList Char -> Int
foo1 XNil  = 0
foo1 (XCons _ t) = 1 + foo1 t
foo2 :: XList () -> Int
foo2 (XListUnit n) = n
```

# Generalised Algebraic Datatypes (GADTs)

Motivationsbeispiel: Mix aus Booleschen und Arithmetische Ausdrücken

```
data Expr = ConstI Int          -- integer constants
          | ConstB Bool         -- boolean constants
          | Or  Expr Expr       -- logic disjunction
          | Add Expr Expr       -- add two expressions
          | Odd Expr            -- convert int to bool
          | If  Expr Expr Expr -- conditional
```

- `(Or (ConstB True) (ConstB False))` ist gültiger Ausdruck
- Auch `If (Odd (Add (ConstI 1) (ConstI 0))) (ConstI 0) (ConstI 1)`.
- `Or (ConstI 1) (ConstB True)` auch **erlaubt**!
- Auswertefunktion für `Expr`: `eval :: Expr -> ???`
- Mögliche Abhilfe: `eval :: Expr -> Either Bool Int`
- Immernoch keine Typsicherheit (z.b. `eval $ Or (ConstB False) (ConstI 69)`)

# Generalised Algebraic Datatypes (GADTs) (2)

Lösung mit zwei getrennten Typen:

```
data BExpr = ConstB Bool | Or BExpr BExpr | Odd IExpr | IfB BExpr BExpr BExpr
data IExpr = ConstI Int  | Add IExpr IExpr | IfI BExpr IExpr IExpr

evalB :: BExpr -> Bool
evalB (ConstB c) = c
evalB (Or a b)   = (evalB a) || (evalB b)
evalB (Odd i)    = odd (evalI i)
evalB (IfB c t e) | evalB c   = evalB t
                  | otherwise = evalB e
evalI :: IExpr -> Int
evalI (ConstI c) = c
evalI (Add a b)  = (evalI a) +  (evalI b)
evalI (IfI c t e) | evalB c   = evalI t
                  | otherwise = evalI e
```

- allerdings keine generische `eval`-Funktion
- und Code für die Auswertung ist verteilt und wechselseitig rekursiv

# Generalised Algebraic Datatypes (GADTs) (3)

Lösung mit Typ-Familien:

```
class ExprClass a where
  data Expr a :: *
  eval :: Expr a -> a

instance ExprClass Bool  where
  data Expr Bool  = ConstB Bool | Or  (Expr Bool) (Expr Bool) | Odd (Expr Int)
                  | IfB (Expr Bool) (Expr Bool) (Expr Bool)
  eval (ConstB c) = c
  eval (Or a b)   = (eval a) || (eval b)
  eval (Odd i)    = odd (eval i)
  eval (IfB c t e) | eval c = eval t
                   | otherwise = eval e

instance ExprClass Int where
  data Expr Int = ConstI Int  | Add (Expr Int)  (Expr Int)
                | IfI (Expr Bool) (Expr Int) (Expr Int)
  eval (ConstI c) = c
  eval (Add a b)  = (eval a) + (eval b)
  eval (IfI c t e) | eval c    = eval t
                   | otherwise = eval e
```

- besser, aber `eval`-Code ist allerdings immer noch verteilt
- Code für `If` ist nicht generisch, da es Wiederholungen für `IfI` und `IfB` gibt.

# Generalised Algebraic Datatypes (GADTs) (4)

```
data Expr = ConstI Int          -- integer constants
          | ConstB Bool         -- boolean constants
          | Or  Expr Expr       -- logic disjunction
          | Add Expr Expr       -- add two expressions
          | Odd Expr            -- convert int to bool
          | If  Expr Expr Expr -- conditional
```

Erinnerung: Typen der Konstruktoren:

```
ConstI :: Int -> Expr
ConstB :: Bool -> Expr
Or     :: Expr -> Expr -> Expr
Add    :: Expr -> Expr -> Expr
Odd    :: Expr -> Expr
If     :: Expr -> Expr -> Expr -> Expr
```

# Generalised Algebraic Datatypes (GADTs) (5)

Mit der Erweiterung `GADTs` (Generalised Algebraic Datatypes):

- Man gibt die Typen direkt an, und
- Verallgemeinerung: Ergebnistyp darf beliebige Instanz des deklarierten Typs sein

Ohne Verallgemeinerung:

```
{-# LANGUAGE GADTs #-}
data Expr where
  ConstI :: Int -> Expr
  ConstB :: Bool -> Expr
  Or     :: Expr -> Expr -> Expr
  Add    :: Expr -> Expr -> Expr
  Odd    :: Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
```

# Generalised Algebraic Datatypes (GADTs) (5)

Trick: Definiere `Expr a` als:

```
{-# LANGUAGE GADTs #-}
data Expr a where
  ConstI :: Int -> Expr Int
  ConstB :: Bool -> Expr Bool
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int -> Expr Int -> Expr Int
  Odd    :: Expr Int -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

- `Or (ConstB False) (ConstI 69)` ist nun typfalsch!
- `If` ist nun generisch und korrekt getypt

# Generalised Algebraic Datatypes (GADTs) (6)

Definition von `eval` mit GADTs:

```
eval :: Expr a -> a
eval (ConstB c) = c
eval (ConstI c) = c
eval (Or  a b)  = (eval a) || (eval b)
eval (Add a b)  = (eval a) +  (eval b)
eval (If c t e) | eval c    = eval t
                | otherwise = eval e
eval (Odd e)    = odd (eval e)
```

- Definition ist nicht erweiterbar
- `deriving` ist für GADTs nicht verwendbar
- Abhilfe: Erweiterung `StandaloneDeriving`

# Generalised Algebraic Datatypes (GADTs) (7)

Mit `StandaloneDeriving`:

```
{-# LANGUAGE GADTs StandaloneDeriving #-}
data Expr a where
  ConstI :: Int -> Expr Int
  ConstB :: Bool -> Expr Bool
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int -> Expr Int -> Expr Int
  Odd    :: Expr Int -> Expr Bool
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
deriving instance (Show a) => Show (Expr a)
deriving instance (Eq a) => Eq (Expr a)
```

# Typfamilien und GADTs

Standardbeispiel: Vektoren

- Vektor = Liste mit *statisch* bekannter Länge
- Ziel: Vektorgröße am Typ erkennbar
- Mit GADTs:

```
{-# LANGUAGE GADTs #-}

data Zero
data Succ n
data Vec size a where -- GADT Syntax
  VecZ :: Vec Zero a
  VecS :: a -> Vec size a -> Vec (Succ size) a
```

- Einen Vektor der Länge 2 erkennt man nun am Typ:

```
v2 :: Vec (Succ (Succ Zero)) Int
v2 = VecS 1 $ VecS 2 $ VecZ
```

# Typfamilien und GADTs (2)

Sortiert Einfügen:

```
insertVec :: (Ord a) => a -> Vec n a -> Vec (Succ n) a
insertVec a     VecZ                 = VecS a VecZ
insertVec a bv@(VecS b v) | a <= b    = VecS a bv
                          | otherwise = VecS b $ insertVec a v
```

Typ zeigt: Vektor wird um 1 länger.

# Typfamilien und GADTs (2)

Sortiert Einfügen:

```
insertVec :: (Ord a) => a -> Vec n a -> Vec (Succ n) a
insertVec a     VecZ                 = VecS a VecZ
insertVec a bv@(VecS b v) | a <= b    = VecS a bv
                          | otherwise = VecS b $ insertVec a v
```

Typ zeigt: Vektor wird um 1 länger.

Sortieren lässt Vektorlänge gleich:

```
isortVec :: (Ord a) => Vec n a -> Vec n a
isortVec VecZ       = VecZ
isortVec (VecS a v) = insertVec a $ isortVec v
```

Compiler (Typcheck) beweist das!

# Typfamilien und GADTs (3)

Problem: Aneinanderhängen zweier Vektoren:

```
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
```

Typ von appendVec???

```
appendVec :: Vec n a -> Vec m a -> Vec n+m a
```

Lösung: Typfamilie

# Typfamilien und GADTs (4)

```
{-# LANGUAGE GADTs TypeFamilies #-}
appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: *
type instance Plus Zero n  = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

# Typfamilien und GADTs (4)

```
{-# LANGUAGE GADTs TypeFamilies #-}
appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: *
type instance Plus Zero n  = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

Alternative: Mit geschlossener Variante der Typfamilie:.

```
{-# LANGUAGE GADTs TypeFamilies #-}
appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: * where
 Plus Zero n  = n
 Plus (Succ m) n = Succ (Plus m n)
```

# DataKinds (1)

Problem:

- kein Zusammenhang zwischen `Zero` und `Succ`
- auch der unsinnige Typ `Vec Bool String` ist erlaubt
- Ursache: Kind von `Vec` ist zu allgemein: `* -> * -> *`
- Wunsch-Kind: `Nat -> * -> *` wobei `Nat` ein neuer Kind ist, der genau die Typen umfasst, die aus `Zero` und `Succ` aufgebaut sind

# DataKinds (2)

Imaginäre Möglichkeit (**so nicht im GHC eingebaut!**):

```
datakind Nat = Zero | Succ Nat
```

- erzeugt Kind `Nat`, der aus den Typkonstruktoren `Zero` (vom Kind `Nat`) und `Succ` vom Kind `Nat -> Nat` aufgebaut ist.
- löst die Probleme!

# DataKinds (3)

Typ und Kind-Definitionen fast gleich:

```
data      Typ = Datenkonstruktor_1 arg_1_1...arg_1_m_1
              | ...
              | Datenkonstruktor_n arg_n_1...arg_n_m_n


-- imaginär (den folgenden Code gibt es nicht)
datakind Kind = Typkonstruktor_1 arg_1_1 ... arg_1_m_1
              | ...
              | Typkonstruktor_n arg_n_1 arg_n_m_n
```

Realität: Erweiterung `DataKinds` erzeugt für jede Datentyp-Definition auch automatisch die Kind-Definition!

# DataKinds (4)

```
{-# LANGUAGE DataKinds #-}
data Bool = True | False
```

- neben dem Typ `Bool` wird auch der Kind `Bool` erzeugt
- neben den Datenkonstruktoren `True` und `False` werden auch die Typen `True` und `False` erzeugt
  (wegen Ambiguität ist die Notation `'True` und `'False` erlaubt)

# DataKinds (5)

Vektorenbeispiel:

```
{-# LANGUAGE GADTs TypeFamilies DataKinds #-}
data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where -- GADT Syntax
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec size a -> Vec ('Succ size) a


appendVec :: Vec n a -> Vec m a -> Vec (Plus n m) a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
type family Plus m n :: Nat where
 Plus 'Zero n  = n
 Plus ('Succ m) n = 'Succ (Plus m n)
```

Im GHCi:

```
*> :t Succ
Succ :: Nat -> Nat
*> :k Succ
Succ :: Nat -> Nat
*> :k 'Succ
'Succ :: Nat -> Nat
```

# DataKinds (6)

- Da es nun unterschiedliche Kinds gibt, gibt es wieder ein Typsystem über Kinds
- Diese Typen heißen Sorte
- Es gibt nur die Sorte `BOX`
- Alle Kinds sind von der Sorte `BOX` (auch `* -> *` usw.)

Problem: Code-Duplikation (programmiere `Plus` auf Typ- und `plus` auf Termebene ...)

Abhilfen (z.B): Dependent Types, singletons-Paket (siehe Literatur)

# Existentielle Typen

- GHC bietet Existentielle Typen mit der Erweiterung `ExistentialQuantification`
- Hauptaufgabe: Typparameter in einer Datentypdefinition „verschwinden“ lassen

- Beispiel:

```
data TaskList tType dType = TaskList {tasks    :: [tType]
                                     ,deadline :: dType  }
```

- Wenn wir `deadline` nur anzeigen lassen wollen uns der `dType` aber egal ist:

```
{-# LANGUAGE ExistentialQuantification #-}
data TaskList tType =
        forall dType. Show dType => TaskList { tasks    :: [tType]
                                             , deadline :: dType  }
```

- Vorteil: Parameter `dType` verschwindet

# Existentielle Typen (2)

```
{-# LANGUAGE ExistentialQuantification #-}
data TaskList tType =
        forall dType. Show dType => TaskList { tasks    :: [tType]
                                             , deadline :: dType  }
```

Quantifizierung legt fest:

- Wir können jeden Typ für `dType` verwenden können, der Instanz der Klasse `Show` ist.
- Wir können nichts anderes mit Feldeintrag `deadline` machen, als es anzeigen zu lassen:

```
-- OK:
showDate :: TaskList a -> String
showDate (TaskList _ dl) = show dl

-- Fehler:
fun (TaskList _ dl) = dl
```

# Existentielle Typen (3)

Quantifizierung ohne Typklassenconstraint:

```
data TaskList tType =
       forall dType.
          TaskList {tasks :: [tType]
                   ,deadline  :: dType}
```

Beliebiger Typ für `dType`, aber man kann mit `deadline` nichts mehr machen!

# Existentielle Typen (4)

Heterogene Datenstrukturen mit existentiellen Typen:

```
data Object = forall a. Show a => Obj a
```

```
*> let list = [Obj 1, Obj True, Obj 'A']
*> :t list
list :: [Object]
*> map (\(Obj s) -> show s) list
["1","True","'A'"]
```

# Existentielle Typen (5)

```
data Object = forall a. Show a => Obj a
```

Quantifizierung: `Obj :: Show a => a -> Object`

- Typ ist ist äquivalent zu `Obj :: forall a. Show a => a -> Object`
  *für jeden Typ* `a` *der Instanz von* `Show` *ist, kann daraus ein* `Object` *erstellt werden.*
- Dies jedoch ebenso äquivalent zu `Obj :: (exists a. Show a => a) -> Object`
  *sobald man einen Typ* `a` *hat, der Instanz von* `Show` *ist, kann man ein* `Object` *erstellen.*

In der Prädikatenlogik: $\forall x.(P(x) \to Q) \equiv (\exists x.P(x)) \to Q$ (falls $x$ nicht in $Q$ vorkommt).

Daher: Schlüsselwort `exists` gibt es nicht in Haskell!

# View Patterns (1)

Gute Praxis:

- Datentypen in Bibliotheken und Modulen abstrakt halten
- Vorteil: Interne Repräsentation änderbar
- Beispiel `Data.Map`, stellt `Map k v` zur Verfügung, aber Implementierung ist versteckt.
- Nachteil: Kein Pattern Matching verfügbar
- Abhilfe: Konvertiere vorher

```
getMinKeyVal :: Map k v -> Maybe (k,v)
getMinKeyVal m = case (toAscList m) of (h:_) -> Just h
                                       []    -> Nothing
```

Anders ausgedrückt:

Eine View-Funktion wie z.B. `toAscList :: Map k v -> [(k,v)]` erlaubt Matching gegen eine Ansicht des Typs `Map k v`

# View Patterns (3)

Spracherweiterung `ViewPatterns` bietet syntaktischen Zucker, um View-Funktionen innerhalb von Pattern Matches einzusetzen:

```
{-# LANGUAGE ViewPatterns #-}
getMinKeyVal :: Map k v -> Maybe (k,v)
getMinKeyVal (toAscList -> (h:_)) = Just h
getMinKeyVal _                    = Nothing
```

Erlaubt Verwendung von Patterns m `(f -> p)` wobei

- `f` ist Funktion, die das zu matchende Argument angewandt wird,
- Ergebnis wird mit Pattern `p` gematched, falls möglich,

# View Patterns (4)

Wenn View-Funktion eine Ausnahme wirft, dann Abbruch!

```
foo (head -> h) = Just h
foo _           = Nothing
```

`foo []` schlägt fehl
Wenn Pattern-Match fehl schlägt, dann wird der nächste, Fall geprüft. Z.B.

```
demo :: Int -> Int -> Int -> Int
demo x (even -> True) z = x+z
demo x y  (even-> True) = x+y
```

```
Prelude> demo 1 3 2
4
Prelude> demo 1 2 3
4
```

# View Patterns (5)

Verschachtelungen von View Patterns:

```
minmax :: [Int] ->  (Int,Int)
minmax (sort -> mi:(reverse -> mx:_)) = (mi,mx)
minmax _ = error "minmax argument size > 1 expected"
```

Allgemeiner: Pattern Guards, da hier beliebige Ausdrücke gematched werden können.

```
getMinKeyValPG :: Map k v -> Maybe (k,v)
getMinKeyValPG m | (h:_) <- toAscList m = Just h
                 | otherwise            = Nothing
```

Aber Pattern-Guards lassen sich nicht verschachteln, daher Zwischenvariablen einführen:

```
minmaxPG :: [Int] ->  (Int,Int)
minmaxPG l | mi:laux <- sort l
           , mx:_    <- reverse laux = (mi,mx)
           | otherwise = error "Argument too small"
```

# Pattern-Synonyme

Erweiterung `PatternSynonyms`: Neue Namen für komplizierte Pattern

Beispiel:

```
data ArithEx = Oper Binop ArithEx ArithEx | Number Int  deriving(Show)
data Binop = AddOp | SubtractOp | TimesOp deriving(Show)

example1 = Oper TimesOp (Oper AddOp (Number 1) (Number 2))
                        (Oper SubtractOp (Number 4) (Number 3))
```

Schöner mit Kombinatoren:

```
myAdd  x y = Oper AddOp x y
mySub  x y = Oper SubtractOp x y
myMult x y = Oper TimesOp x y

example2 = myMult (myAdd (Number 1) (Number 2))
                  (mySub (Number 4) (Number 3))
```

# Pattern-Synonyme (2)

Kombinatoren: Funktionieren nur bei der Konstruktion, nicht beim Zerlegen

```
exec (Oper AddOp x y)      = (exec x) + (exec y)
exec (Oper TimesOp x y)    = (exec x) * (exec y)
exec (Oper SubtractOp x y) = (exec x) - (exec y)
exec (Number x)            = x
```

Mit Synonymen

```
{-# LANGUAGE PatternSynonyms #-}
pattern MyAdd  x y  = Oper AddOp x y
pattern MySub  x y  = Oper SubtractOp x y
pattern MyMult x y  = Oper TimesOp x y

exec' (MyAdd x y)  = (exec' x) + (exec' y)
exec' (MyMult x y) = (exec' x) * (exec' y)
exec' (MySub x y)  = (exec' x) - (exec' y)
exec' (Number x)   = x
```

# Pattern-Synonyme (3)

```
{-# LANGUAGE PatternSynonyms #-}
pattern MyAdd  x y  = Oper AddOp x y
pattern MySub  x y  = Oper SubtractOp x y
pattern MyMult x y  = Oper TimesOp x y
```

Damit sind auch Synonyme für Konstrukoren definiert, daher funktioniert:

```
example3 = MyMult
             (MyAdd (Number 1) (Number 2))
             (MySub (Number 4) (Number 3))
```

# Pattern-Synonyme (4)

Unidirektionale Pattern: Erlauben nur das Verwenden in Pattern.

Verwende `pattern ... <- ...` statt `pattern ... = ...`

Beipiel:

```
pattern Fuenftes x <- (_:_:_:_:x:_)

getFuenftes :: [a] -> Maybe a
getFuenftes (Fuenftes x) = Just x
getFuenftes _            = Nothing
```

Z.B. `getFuenftes "ABCDEFG"` $\rightarrow$ `'E'`,

Aber `Fuenftes 'E'` würde keinen Sinn ergeben

# Überladene Strings

- In Haskell ist `"Burp"` immer vom Typ `String = [Char]`
- Es gibt aber effizientere Repräsentationen,
  z.B. `Data.ByteString` für 8-Bit-Arrays und `Data.Text` für Unicode-Strings.
- Nachteil: Konvertiertierung nötig:

```
pack   :: String -> Text
unpack :: Text   -> String
```

- Literale wie `9` können jedoch verschiedene Typen haben, z.B. `Int` oder `Double`.
- Die Spracherweiterung `OverloadedStrings` erlaubt das Gleiche auch für String-Literale.

# Überladene Strings (2)

Typ muss Instanz der Klasse `IsString` aus Modul `Data.String` sein:

```
class IsString a where
   fromString :: String -> a
```

String-Literale haben dann den Typ `(IsString a) => a`,
d.h. Konvertierung erfolgt implizit.

```
Prelude> :t "Hallo"
"Hallo" :: [Char]
Prelude> :set -XOverloadedStrings
Prelude> :t "Hallo"
"Hallo" :: Data.String.IsString p => p
```

# Überladene Strings (3)

Nachteile: Machmal Typannotationen notwendig

Künstliches (nicht sinnvolles) Beispiel:

```
ghci -XOverloadedStrings
*> :module + Data.String
*> instance IsString Bool where fromString "True" = True;
                                fromString   _    = False
*> instance IsString Int  where fromString s = length s
*> fromString "True"
"True"
*> fromString "True" :: Bool
True
*> fromString "True" :: Int
4
```

# Überladene Strings: Größeres Beispiel

```
{-# LANGUAGE OverloadedStrings #-}
import Data.String
newtype MyString = MyString String   deriving (Eq)

instance IsString MyString where
    fromString = MyString
instance Show MyString where
    show (MyString s) = "<" ++ s ++ ">"

greet :: MyString -> MyString
greet (MyString "hello") = MyString "hallo"
greet "fool" = "welt"     -- 2x automatische Konvertierung,
greet other  = other      -- also auch im Pattern-Match!

main = do
  print $ greet "hello"   -- automatische Konvertierung!
  print $ greet "fool"    -- automatische Konvertierung!
```