Fortgeschrittene Funktionale Programmierung

Teil 8: Spracherweiterungen

Steffen Jost

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

21. Januar 2019

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 8: Spracherweiterungen

1. IORef
2. MultiParamTypeClasses
3. TypeFamilies I
4. Generalised Algebraic Datatypes
5. TypeFamilies II
6. View Patterns
7. OverloadedStrings

GHC Spracherweiterungen

Das Yesod-Framework für Web-Applikationen verwendet zahlreiche Spracherweiterungen von GHC.
Diese Spracherweiterungen sind natürlich auch unabhängig von der Anwendung in Yesod von Interesse, weshalb wir diese nun zuerst in Isolation betrachten.

Der erste Abschnitt **IORef** (8.4ff.) benötigt keinerlei Spracherweiterung, passt aber als Motivation hier ebenfalls gut hinein.

Veränderliche Variablen

`IO`-Monade erlaubt echt-veränderliche Variablen

Module `Data.IORef` definiert:

```
newIORef    :: a -> IO (IORef a)          Referenz erzeugen
readIORef   :: IORef a -> IO a            Referenz auslesen
writeIORef  :: IORef a -> a -> IO ()      Referenz schreiben
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

Referenz bearbeiten, strikte Variante: `modifyIORef'`

Einfachste Art von Variablen, funktioniert wie `MVar`, `TVar` aber:

- Operationen sind nicht atomar!
  $\Rightarrow$ Immer nur im gleichen Thread verwenden
- Operationen können in anderer Reihenfolge ablaufen!
  $\Rightarrow$ Nicht für Locks einsetzen, sondern `MVar` einsetzen

Multi-Parameter Typklassen

Erweiterung `{-# LANGUAGE MultiParamTypeClasses #-}` erlaubt Typklassen mit mehreren Parametern:

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

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

Funktion `addOne` kann mit `IORef`, `TVar` oder `MVar` arbeiten!

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

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

Multi-Parameter Typklassen: Beispiel

```
class Monad m => VarMonad m v where ...
instance VarMonad IO  IORef   where ...
instance VarMonad IO  MVar    where ...
instance VarMonad STM TVar    where ...
addOne  :: (VarMonad m v, Num a) => v a -> m ()

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"
```

Probleme:

1. Typ-Inferenz wird deutlicher komplizierter und teilweise unklar

Multi-Parameter Typklassen: Beispiel

```
class Monad m => VarMonad m v where ...
instance VarMonad IO  IORef   where ...
instance VarMonad IO  MVar    where ...
instance VarMonad STM TVar    where ...
addOne  :: (VarMonad m v, Num a) => v a -> m ()

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"
```

Probleme:

2. 2. Parameter `v` hängt von `m` ab, z.B. `IO` und `TVar` geht nicht.
   Lösung: **Functional Dependencies** `m -> v` oder Typ-Familien

Typ-Familien

Bekannt:

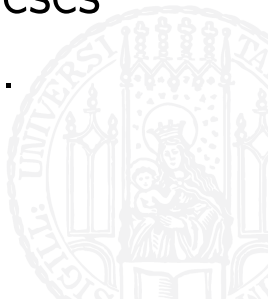
Überladen von Funktionen für mehrere Datentypen mit Typklassen.

Neu:

Erweiterung `{-# LANGUAGE TypeFamilies #-}` für Typ-Familien erlaubt das **Überladen von Datentypen** selbst.

Typ-Familien sind eine sehr mächtige, ausdrucksstarke Erweiterung des Typsystems, welche bereits **Funktionen auf Typ-Ebene** gestatten.

Für die reine Verwendung des Yesod Frameworks müssen wir dieses Thema nicht vertiefen; wir beschränken uns daher auf Beispiele.



```
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
```

```
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 m` berechnet Typ der Referenz

Jetzt können wir aber nur noch eine Instanz für `IO` angeben (entweder `IORef` oder `MVar`).

Beispielmöglichkeit Typ-Familien

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

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 Typ Familien

Wir betrachten bis jetzt nur mit Klassen **assoziierte Typ-Familien**. Dies ist jedoch keine Notwendigkeit:

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

Hier haben wir eine Funktion `G` auf Typ-Ebene, welche den Typ `Int` auf den Typ `Bool` abbildet und alle anderen Typen auf `Char`.

Das obige Beispiel ist eine **geschlossene** Deklaration. **Offene**, d.h. erweiterbar Deklarationen sind ebenfalls möglich:

```
type family F a b :: * -> *
type instance F Int Bool = Char
```


$\Rightarrow$ In beiden Fällen werden lediglich Typ-Synonyme definiert.

Datentyp Familien

Die Erweiterung `TypeFamilies` erlaubt nicht nur Typ-Synonyme, sondern auch **offene Datentypdeklarationen**; wiederum entweder assoziiert mit Typklassen oder wie hier auf dem Top-Level:

```
data family XList a

data instance XList Char = XCons !Char !(XList Char) | XNil
newtype instance XList () = XListUnit Int
```

Hier wird ein Datentyp für Listen deklariert, der für `Char` strikt arbeitet und für Unit-Listen, d.h. Listen über Typ `()`, effizient durch eine ganze Zahl approximiert wird.

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

foo (XListUnit n) = n
```

Datentyp Familien

Die Erweiterung `TypeFamilies` erlaubt nicht nur Typ-Synonyme, sondern auch **offene Datentypdeklarationen**; wiederum entweder assoziiert mit Typklassen oder wie hier auf dem Top-Level:

```
data family XList a

data instance XList Char = XCons !Char !(XList Char) | XNil
newtype instance XList () = XListUnit Int
```

Hier wird ein Datentyp für Listen deklariert, der für `Char` strikt arbeitet und für Unit-Listen, d.h. Listen über Typ `()`, effizient durch eine ganze Zahl approximiert wird.

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

foo (XListUnit n) = n
```

Datentyp Familien sind offen, d.h. Deklaration `foo` wäre möglicherweise unvollständig. Durchreichen von `XList a` ist aber möglich.

Datentyp Familien

Die Erweiterung `TypeFamilies` erlaubt nicht nur Typ-Synonyme, sondern auch **offene Datentypdeklarationen**; wiederum entweder assoziiert mit Typklassen oder wie hier auf dem Top-Level:

```
data family XList a

data instance XList Char = XCons !Char !(XList Char) | XNil
newtype instance XList () = XListUnit Int
```

Hier wird ein Datentyp für Listen deklariert, der für `Char` strikt arbeitet und für Unit-Listen, d.h. Listen über Typ `()`, effizient durch eine ganze Zahl approximiert wird.

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

Datentyp Familien

Die Erweiterung `TypeFamilies` erlaubt nicht nur Typ-Synonyme, sondern auch **offene Datentypdeklarationen**; wiederum entweder assoziiert mit Typklassen oder wie hier auf dem Top-Level:

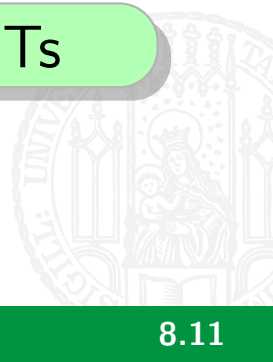
```
data family XList a

data instance XList Char = XCons !Char !(XList Char) | XNil
newtype instance XList () = XListUnit Int
```

Hier wird ein Datentyp für Listen deklariert, der für `Char` strikt arbeitet und für Unit-Listen, d.h. Listen über Typ `()`, effizient durch eine ganze Zahl approximiert wird.

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

Alternative: GADTs



Problem mit gewöhnlichen Datentypen

```
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
```

Welchen Ergebnistyp hätte eine Auswertefunktion?

```
eval :: Expr -> ???
```

Es kommen sowohl `Bool` als auch `Int` in Frage.

Hier könnte `eval :: Expr -> Either Bool Int` aushelfen, aber dabei entstehen neue Probleme:

```
eval $ Or (ConstB False) (ConstI 69)
```

$\Rightarrow$ Keine Typsicherheit innerhalb der Datenstruktur!

Problem mit gewöhnlichen Datentypen

```
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
```

Welchen Ergebnistyp hätte eine Auswertefunktion?

```
eval :: Expr -> ???
```

Es kommen sowohl `Bool` als auch `Int` in Frage.

Hier könnte `eval :: Expr -> Either Bool Int` aushelfen, aber dabei entstehen neue Probleme:

```
eval $ Or (ConstB False) (ConstI 69)
```

$\Rightarrow$ Keine Typsicherheit innerhalb der Datenstruktur!

Lösung 1: Getrennte Typen

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

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
```

- `eval` Funktion nicht generisch
- Code für `eval` verteilt, wechselseitig rekursiv

Lösung 2: 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)
  eval (ConstB c) = c
  eval (Or a b)   = (eval a) || (eval b)
  eval (Odd i)    = odd (eval i)

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
```

- `eval`-Code immer noch verteilt
- `If` nicht generisch, d.h. Wiederholungen für `IfI`, `IfB`,...

Typ eines Konstruktors

Erinnerung: Konstruktoren können als Funktionen aufgefasst werden
Konstruktoren berechnen nichts, vermerken Argumente im Speicher

Zum Beispiel für den ursprünglichen Typ gilt:

```
data Expr = ConstB Bool    | ConstI Int
          | Or  Expr Expr | Add Expr Expr
          | Odd Expr | If  Expr Expr Expr
```

```
> :t ConstB
ConstB :: Bool -> Expr
> :t Add
Add :: Expr -> Expr -> Expr
> :t If
If :: Expr -> Expr -> Expr -> Expr
```

Generalised Algebraic Datatypes (GADTs)

Erweiterung **Generalised Algebraic Datatypes** verallgemeinert:

- Konstruktoren werden jetzt durch Ihren Typ beschrieben
- *Ergebnistyp* darf *beliebige Instanz* des deklarierten Typen sein

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

- Funktion `eval :: Expr a -> a` *typsicher* definierbar
- Generisches, typsicheres `If` möglich
- Im Gegensatz zu Typ Familien nicht erweiterbar, d.h. alle Definition müssen am gleichen Ort sein
- `deriving` nur für ADTs in GADT Syntax möglich

Lösung 3: GADTs

Pattern Matching funktioniert wie gewohnt:

```
data Expr a where
  ConstB :: Bool -> Expr Bool
  ConstI :: Int  -> Expr Int
  Or     :: Expr Bool -> Expr Bool -> Expr Bool
  Add    :: Expr Int  -> Expr Int  -> Expr Int
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a

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
```

- GADTs werden primär für *typsichere DSLs* verwendet

Vektoren mit statischer Grösse

Das Standardbeispiel für eigenständige Typ-Familien ist durch Vektoren motiviert, wobei wir hier unter einem **Vektor** eine Listen mit *statisch* bekannter Länge interpretieren:

D.h. Vektoren verschiedener Größe haben verschiedene Typen

```
{-# LANGUAGE GADTs, KindSignatures #-}
data Zero; data Succ n
data Vec :: * -> * -> * where -- GADT Syntax
  VecZ :: Vec Zero a
  VecS :: a -> Vec n a -> Vec (Succ n) a

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

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
```

Vektoren mit statischer Grösse

Das Standardbeispiel für eigenständige Typ-Familien ist durch Vektoren motiviert, wobei wir hier unter einem **Vektor** eine Listen mit *statisch* bekannter Länge interpretieren:

D.h. Vektoren verschiedener Größe haben verschiedene Typen

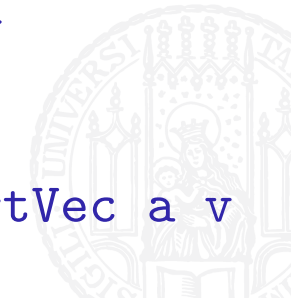
```
{-# LANGUAGE GADTs, KindSignatures #-}
data Zero; data Succ n
data Vec :: * -> * -> * where -- GADT Syn
  VecZ :: Vec Zero a
  VecS :: a -> Vec n a -> Vec (Succ n) a

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

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
```

Kinds:

```
Zero :: *
Succ :: * -> *
Vec  :: * -> * -> *
```



Type Synonym Familien

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sortieren eines Vektors dessen Größe unverändert läßt:

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

Doch welchen Typ hat die Verkettung zweier Vektoren?

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

Type Synonym Familien

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sortieren eines Vektors dessen Größe unverändert läßt:

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

Doch welchen Typ hat die Verkettung zweier Vektoren?

```
appendVec :: Vec n a -> Vec m a -> Vec ??? a
appendVec  VecZ      v = v
appendVec (VecS a w) v = VecS a (appendVec w v)
```

Type Synonym Familien

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sortieren eines Vektors dessen Größe unverändert läßt:

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

Doch welchen Typ hat die Verkettung zweier Vektoren?

```
{-# 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 :: *              -- OPEN (erweiterbar)
type instance Plus   Zero   n = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

Type Synonym Familie definiert Abbildung auf Typ-Ebene.

Type Synonym Familien

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sortieren eines Vektors dessen Größe unverändert läßt:

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

Doch welchen Typ hat die Verkettung zweier Vektoren?

```
{-# 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    -- CLOSED (nicht erw.)
  Plus   Zero   n = n
  Plus (Succ n) m = Succ (Plus n m)
```

Type Synonym Familie definiert Abbildung auf Typ-Ebene.

Type Synonym Familien

Mit dem gezeigten Code können wir vom Compiler überprüfen lassen, dass Sor

```
isortVec ::
isortVec Vec
isortVec (Ve
```

Doch welchen T

```
{-# LANGUAGE
appendVec :
appendVec  V
appendVec (V
```

- Typ Synonym Familien:
  `type family...`
  `type instance...`
  Instanzen liefern bereits definierte Typen
- Datentyp Familien:
  `data family NewT t :: *...`
  `data instance...`
  Instanzen liefern frischen Typ `NewT :: * -> *`
  (u.a. vereinfacht Type-Checking)

```
type family Plus m n :: *                    -- OPEN (erweiterbar)
type instance Plus   Zero    n = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

Type Synonym Familie definiert Abbildung auf Typ-Ebene.

Erweiterung: Data Kinds

Probleme

- `Zero` und `Succ` haben nichts miteinander zu tun!
- `Vec Bool String` unsinnig, aber nicht verboten.

Lösung

```
{-# LANGUAGE GADTs, TypeFamilies, DataKinds #-}

data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where                    -- GADT Syntax
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec n a -> Vec ('Succ n) a

type family Plus n m :: Nat where                  -- CLOSED
  Plus 'Zero n = n
  Plus ('Succ n) m = 'Succ (Plus n m)
```

Erweiterung: Data Kinds

Data Kinds Promotion für Datentypen

- Konstruktoren `Zero` und `Succ` werden befördert zu zusätzlichen Typen `'Zero` und `'Succ`

  Apostrophen sind optional, falls eindeutig

- Typ `Nat` wird zu Kind `Nat`
- Kind `Nat` und `*` haben beit Sort `BOX`

```
{-# LANGUAGE GADTs, TypeFamilies, DataKinds #-}

data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec n a -> Vec ('Succ n) a

type family Plus n m :: Nat where
  Plus 'Zero n = n
  Plus ('Succ n) m = 'Succ (Plus n m)
```

```
> :set -XDataKinds
> :type Succ
Succ :: Nat -> Nat
> :kind 'Succ
'Succ :: Nat -> Nat
> :kind Plus
Vec :: Nat->Nat->Nat
```

Erweiterung: Data Kinds

Problem Sinnlose Code Duplikation zwischen `plus` und `Plus`

```
plus :: Nat -> Nat -> Nat
plus Zero n  = n
plus (Succ n) m = Succ $ n `plus` m
```

```
{-# LANGUAGE GADTs, TypeFamilies, DataKinds #-}

data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec n a -> Vec ('Succ n) a

type family Plus n m :: Nat where
  Plus 'Zero n = n
  Plus ('Succ n) m = 'Succ (Plus n m)
```

```
> :set -XDataKinds
> :type Succ
Succ :: Nat -> Nat
> :kind 'Succ
'Succ :: Nat -> Nat
> :kind Plus
Vec :: Nat->Nat->Nat
```

Erweiterung: Data Kinds

Problem Sinnlose Code Duplikation zwischen `plus` und `Plus`

```
plus :: Nat -> Nat -> Nat
plus Zero n  = n
plus (Succ n) m = Succ $ n `plus` m
```

Lösung I: Dependent Types, d.h. es werden Werte werden innerhalb von Typen erlaubt Agda, Idris, Coq

```
{-# LANGUAGE GADTs, TypeFamilies, DataKinds #-}

data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec n a -> Vec ('Succ n) a

type family Plus n m :: Nat where
  Plus 'Zero n = n
  Plus ('Succ n) m = 'Succ (Plus n m)
```

```
> :set -XDataKinds
> :type Succ
Succ :: Nat -> Nat
> :kind 'Succ
'Succ :: Nat -> Nat
> :kind Plus
Vec :: Nat->Nat->Nat
```

Erweiterung: Data Kinds

Problem Sinnlose Code Duplikation zwischen `plus` und `Plus`

```
plus :: Nat -> Nat -> Nat
plus Zero n  = n
plus (Succ n) m = Succ $ n `plus` m
```

Lösung II: Singleton Types, siehe `Data.Singletons`
Jeder Typ wird assoziert mit neuem Typ mit einzelnen Wert.

```
{-# LANGUAGE GADTs, TypeFamilies, DataKinds #-}

data Nat = Zero | Succ Nat
data Vec :: Nat -> * -> * where
  VecZ :: Vec 'Zero a
  VecS :: a -> Vec n a -> Vec ('Succ n) a

type family Plus n m :: Nat where
  Plus 'Zero n = n
  Plus ('Succ n) m = 'Succ (Plus n m)
```

```
> :set -XDataKinds
> :type Succ
Succ :: Nat -> Nat
> :kind 'Succ
'Succ :: Nat -> Nat
> :kind Plus
Vec :: Nat->Nat->Nat
```

Dependent Types

Lösung I:

Dependent Types, d.h. es werden Werte werden innerhalb von Typen erlaubt Agda, Idris, Coq

Typsystem muss dann in der Lage sein, Dinge wie `Plus v Zero = v` zu inferieren, d.h. das Typsystem

- Beweisassistenten
- Voller Beweisassistent
- Typen müssen zur Laufzeit bekannt sein
- Beweisbare Terminierung oft gefordert

Lösung II: Singleton Types, siehe `Data.Singletons`

Jeder Typ wird assoziiert mit neuem Typ mit einzelnen Wert.

- Viele, viele neue Typen und Kinds; können aber mit Template Haskell automatisch generiert werden
- Mit rückwärts-kompatiblen Spracherweiterungen innerhalb von GHC bereits verfügbar

Matching vs Abstraktion

Es ist gute Praxis, Datentypen (in Modulen) abstrakt zu halten, z.B. um später gefahrlos die Repräsentation zu ändern.

Beispiel

Modul `Data.Map` stellt **abstrakten Datentyp** `Map k v` zur Verfügung, die Implementierung ist jedoch versteckt.
Ausschließlich bereitgestellte Funktionen verwendbar $\Rightarrow$ Schnittstelle

Nachteil

Das mächtige Werkzeug **Pattern Matching** können wir mit dem Typ `Map k v` nur noch *indirekt* einsetzen:

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

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

View Patterns

Spracherweiterung **View Patterns** 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
```

Spracherweiterung `ViewPatterns` erlaubt allgemein die Verwendung von Patterns der Form `(f -> p)`

- `f` ist eine Funktion, welche auf das zu matchende Argument angewandt wird
- Ergebnis wird mit Pattern `p` gematched, falls möglich
- Dieses Pattern schlägt fehl, wenn der Match mit dem *Ergebnis* der Funktionsanwendung nicht gelingt

View Patterns

Schlägt der anschliessende Pattern-Match fehl, dann wird einfach der nächsten Fall geprüft:

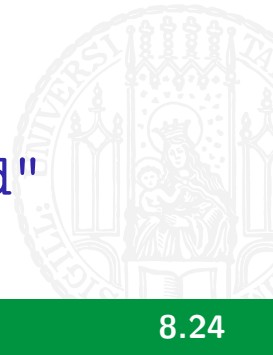
```
demo :: Int -> Int -> Int -> Int
demo x (even -> True) z = x+z
demo x y  (even-> True) = x+y
```

Fallstrick: Wirft die View-Funktion eine Ausnahme, dann wird komplett abgebrochen, ohne andere Fälle zu prüfen

```
doesntwork (head -> h) = h
doesntwork []          = 0 -- never tested, change order
```

Verschachtelungen von View Patterns sind erlaubt:

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



View Patterns vs Pattern Guards

Eine noch allgemeinere Alternative bieten die bereits behandelten 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
```

Pattern-Guards lassen sich jedoch nicht derart verschachteln, weshalb man hier Zwischenvariablen einführen muss:

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

Strings in Haskell

Der Ausdruck `"Burp"` kann nur Typ `String` haben,
wobei `String` ein Typsynonym für `[Char]` ist

Typ `[Char]` ist leicht verständlich, aber sehr ineffizient!
Besser Alternativen z.B. durch Module

- `Data.ByteString` für 8-Bit Arrays
- `Data.Text` für Unicode Strings

Nachteil

Jeder String im Quellcode muss erst umständlich in den anderen Typ konvertiert werden:

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

Außerdem tauchen zur Laufzeit wieder gewöhnliche Strings im Speicher auf.

OverloadedStrings

Literal `9` kann verschiedene Typen haben, z.B. `Int` oder `Double`.

Spracherweiterung `{-# LANGUAGE OverloadedStrings #-}` erlaubt das Gleiche auch für String-Literale.

Voraussetzung

Gewünschter 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.

Nachteile

- Typannotationen notwendig, falls mehrere Instanzen der Klasse `IsString` in Frage kommen
- Fehlermeldungen können komplizierter aussehen

Typ bestimmt Ergebnis

Der Typparameter taucht nur im Ergebnis auf:

```
fromString :: String -> a
```

d.h. der benötigte Ergebnistyp bestimmt den verwendeten Code

ebenso bei `read` aus Klasse `Read`

```
ghci
> :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
```

Typ bestimmt Ergebnis

Der Typparameter taucht nur im Ergebnis auf:

```
fromString :: String -> a
```

d.h. der benötigte Ergebnistyp bestimmt den verwendeten Code

ebenso bei `read` aus Klasse `Read`

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

Typ bestimmt Ergebnis

Der Typparameter taucht nur im Ergebnis auf:

```
fromString :: String -> a
```

d.h. der benötigte Ergebnistyp bestimmt den verwendeten Code

ebenso bei `read` aus Klasse `Read`

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

WARNUNG: Unsinniges Beispiel!

Nur zur Demonstration der prinzipiellen Funktionsweise.

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!
```

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!
```

Ausgabe main:
<hallo>
<welt>