Fortgeschrittene Funktionale Programmierung

Teil 8: Webapplikationen mit Yesod

Steffen Jost

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

4. Juli 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 8: Webapplikationen mit Yesod

1. Grundlagen
   - OverloadedStrings
   - MultiParamTypeClasses
     - IORef
   - TypeFamilies
   - Generalised Algebraic Datatypes
   - View Patterns
   - TemplateHaskell
2. Shakespeare
   - Hamlet
   - Lucius
   - Cassius
   - Julius
3. Yesod
   - Widgets
   - Type Families Revisited
   - Foundation Type
   - Routing
   - Handling
   - Formulare
   - Sessions
     - Messages
     - Ultimate Destination
4. Persistent
   - Spezifizieren
   - Migration
   - Zugriff
   - Integration in Yesod
5. Weitere Themen

Web-App Frameworks für Haskell

Zur Entwicklung von **Webapplikation** bietet sich die Verwendung eines Frameworks an, welches alle grundlegenden Funktionalitäten durch Bibliotheken und vordefinierte Programmgerüste bereitgestellt.

Die wichtigsten Web Development Frameworks für Haskell sind:
Happstack, Yesod, Snap and Servant — mix & match möglich!

Im folgenden betrachten wir Yesod 1.4

- Typ-sichere URLs — generell viele statische Prüfungen
- Templating / DSLs — viele modulare Einzelteile
- integrierte Datenbankanbindung — `persist / conduit`
- REST Architektur — Representational State Transfer
  zustandslos, d.h. gleiche URL = gleiche Webseite

Yesod Installation

Installation mit `cabal install` möglich, aber wegen vieler Abhängigkeiten oft schwierig. Installation über `stack` empfohlen:

```
> stack new mein-projekt yesod-sqlite
> cd mein-projekt
> stack build yesod-bin cabal-install --install-ghc
> stack build
> stack exec -- yesod devel
```

Folgende Webseite sollte dann *lokal* abrufbar sein:
http://localhost:3000/

Weitere Information zur Installation von Alex 2016 zusammengestellt auf:

http://github.com/cirquit/ffp-lib/blob/master/docs/stack-n-yesod.md

Scaffolding Tool

Stack bringt Templates (Grundgerüste) mit:

```
stack new my-project yesod-sqlite
```

Es gibt weitere Templates, anzeigen mit: `stack templates`

Die Gerüste haben zahlreiche nicht-zwingende, aber sinnvolle Voreinstellung, z.B. sind viele Dinge in mehreren Dateien getrennt, was nicht unbedingt notwendig ist. Beispiele hier oft in einer Datei.

```
stack exec -- yesod devel
```

startet Development Webserver auf http://localhost:3000

Eventuell Umgebungsvariablen `HOST` und `PORT` einstellen

Nicht von allen Templates unterstützt

- `yesod devel` überwacht den Quellcode; ggf. wird der Development Webserver kompiliert & neugestartet.
- Compiler-Optionen, Paket-Abhängigkeiten in `package.yaml` einstellen cabal-Datei wird von `hpack`-Tool erstellt

Hello Yesod

```haskell
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```haskell
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

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
> :module + Data.String
> instance IsString Bool where fromString "True" = True;
                                String    _      = False
> in                            String s = length s
> fr
"Tru
> fromString "True" :: Bool
True
> fromString "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>
```

Veränderliche Variablen

`IO`-Monade erlaubt echt-veränderliche Variablen

Module `Data.IORef` definiert:

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

- `newIORef`: Referenz erzeugen
- `readIORef`: Referenz auslesen
- `writeIORef`: Referenz schreiben
- `modifyIORef`: Referenz bearbeiten, strikte Variante: `modifyIORef'`

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

- Operationen sind nicht atomar!
  $\Rightarrow$ Nur in einem 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:

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

Typ-Familien

Typklassen das Überladen von Funktionen für mehrere Datentypen.

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 jedoch 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
```

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
          | 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)
```

⇒ 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
          | 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)
```

⇒ Keine Typsicherheit innerhalb der Datenstruktur!

Lösung 1: Getrennte Typen

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

evalB :: BExpr -> Bool
evalB (ConstB c) = c
evalB (Or a b)   = (evalB a) || (evalB b)

evalI :: IExpr -> Int
evalI (ConstI c) = c
evalI (Add a b)  = (evalI a) +  (evalI b)
evalI (If 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)
  eval (ConstB c) = c
  eval (Or a b)   = (eval a) || (eval b)

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
          | 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
  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

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 ⇒ 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 `toAscList :: Map k v -> [(k,v)]` erlaubt uns *Ansicht* auf Typ `Map k v`, gegen die wir wieder matchen können.

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 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"
```

Metaprogrammierung

Manchmal möchte man den Quelltext eines Programms nicht direkt selbst programmieren, sondern durch ein Programm bearbeiten lassen.

Weit verbreitet ist dazu der C-Präprozessor:

```
#define VERSION 2


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

Template Haskell geht darüber noch 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.

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.

Diese Wechsel der Ebenen dürfen auch verschachtelt werden!

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

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)

Für Yesod benötigen wir vor allem letzteres, d.h. zur Manipulation von HTML oder JavaScript.
Andere Aspekte von Template Haskell stellen wir momentan zurück.

Template Haskell Beispiele

Genererische 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') )
```

Hello Yesod

```
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Yesod Templates

HTML, CSS und JavaScript werden mit Templates programmiert.
Template Haskell für diese Sprachen erlaubt die Code Manipulation:

```
[whamlet|
  <h2>Hello Yesod!
  Some text that is <i>displayed</i> here.
|]
```

Templates können direkt per Quasi-Quote oder aus separaten Dateien erstellt werden:

Sprache	*Quasiquoter*	*Dateiendung*
HTML	`hamlet`	.hamlet
CSS	`cassius`	.cassius
CSS	`lucius`	.lucius
JavaScript	`julius`	.julius
Interpolierter Text	`stext/ltext`	

Spezialisierte Varianten `shamlet`, `whamlet`, ...
unterscheiden sich meist nur durch Ergebnistyp.

Yesod Templates und Interpolation

Entsprechend dem Spleißen für Haskell-Code werden Templates durch **Interpolation** eingebunden:

- `#{ }` Interpolation für Variablen im Scope (escaped)
- `@{ }` Typsichere URL Interpolation, also `@{HomeR}`
- `^{ }` Template embedding, fügt Template gleichen Typs ein

Damit können wir Templates dynamisch gestalten:

```
let foo = fib 22 in
...
[whamlet|
  Value of foo is #{foo}
|]
```

Ergebnistyp einer Interpolation muss immer eine Instanz der Typklasse `ToHtml` sein   funktioniert ganz analog zu `Show`

Fallstrick: Typ von `foo` muss inferierbar sein, ggf. Typ angegeben

Variablen Interpolation

Interpolation von URLs funktioniert ähnlich:

```
let foo = fib 22 in [whamlet|
  Value of foo is #{foo}

  Return to <a href=@{Home}>Homepage
  .
|]
```

Dabei muss `Home` ein Konstruktor/Wert des Datentyps für das *Routing* dieser Webanwendung sein, welches wir später genauer betrachten werden.

Hamlet I

Hamlet funktioniert wie gewöhnliches HTML plus Interpolation

Zusätzlich gilt:

- Schließende HTML-Tags werden durch Einrücken ersetzt:

```
<p>Some paragraph.
  <ul><li>Item 1</li>
      <li>Item 2</li></ul></p>
<p>Next paragraph.</p>
```

  dieses HTML wird in Hamlet so geschrieben:

```
<p>Some paragraph.
  <ul>
    <li>Item 1
    <li>Item 2
<p>Some paragraph.
```

  Quasiquoter generiert oberen Code aus dem unteren.

Hamlet II

Zusätzlich gilt:

- Kurze geschlossene inline Tags sind zulässig:

  ```
  <p>Some <i>italic</i> paragraph.
  ```

  Wichtig: Zeile darf nicht mit Tag beginnen, da ansonsten Einrückungen erwartet wird.

- Leerzeichen am Zeilenanfang und -ende, so wie vor und nach Tags müssen mit # und \ markiert werden:

  ```
  <p>
    Some #
    <i>italic
    \ paragraph.
  ```

Hamlet III

Zusätzlich gilt:

- Attribute funktionieren wie in HTML, d.h. Gleichheitszeichen, Wert und Anführungszeichen sind meist optional.

  Abkürzungen für IDs, Klassen und Konditionale erlaubt:

```
<p #paragraphid .class1 .class2>
<p :someBool:style="color:red">
<input type=checkbox :isChecked:checked>
```

- Ein Attribut-Paar `attr::(Text,Text)` oder mehrere `attrs::[(Text,Text)]` können auch direkt eingebunden werden:

```
<p *{attrs}>
```

Hamlet IV

Hamlet erlaubt auch logische Konstrukte

- Konditional

```
$if isAdmin
  <p>Hallo mein Administrator!
$elseif isLoggedIn
  <p>Du bist nicht mein Administrator.
$else
  <p>Wer bist Du?
```

- Einfache Schleifen:

```
$if null people
  <p>Niemand registriert.
$else
  <ul>
    $forall person <- people
      <li>#{person}
```

Hamlet V

- Maybe & einfaches Pattern-Matching

```
$maybe name <- maybeName
  <p>Dein Name ist #{name}
$nothing
  <p>Ich kenne Dich nicht.

$maybe Person vorname nachname <- maybePerson
  <p> Dein Name ist #{vorname} #{nachname}
```

- Volles Pattern-Matching mit Case

```
$case foo
  $of Left bar
      <p>Dies war links: #{bar}
  $of Right baz
      <p>Dies war rechts: #{baz}
```

Hamlet VI

- `$with` ist das neue `let`, also für lokale Definitionen

```
$with foo <- myfun argument $ otherfun more args
  <p>
    Einmal ausgewertetes foo hier #{foo}
    und da #{foo} und dort #{foo} verwendet.
```

- Abkürzungen für Standard Komponenten vordefiniert:

```
$doctype 5
```

steht zum Beispiel für

```
<!DOCTYPE html>
```

Lucius

Lucius akzeptiert ganz normales CSS. Zusätzlich ist erlaubt

- Interpolation für Variablen `#{}`, URLs `@{}` und Mixins `^{}`
- CSS Blöcke dürfen verschachtelt werden
- Es können lokale Variablen deklariert werden

Beispiel:

```
article code { background-color: grey; }
article p { text-indent: 2em; }
article a { text-decoration: none; }
```

kann bei Bedarf umgeschrieben werden zu

```
@backgroundcolor: grey;
article {
  code { background-color: #{backgroundcolor}; }
  p { text-indent: 2em; }
  a { text-decoration: none; }
```

Cassius

Eignet sich für Whitespace-sensitive Haskell-Programmierer als Alternative zu Lucius:

Cassius wird zu Lucius übersetzt. Klammern und Semikolon *müssen* immer durch Einrücken ersetzt werden:

```
#banner
  border: 1px solid #{bannerColor}
  background-image: url(@{BannerImageR})
```

⇒ Für vorhandenen CSS Code immer Lucius einsetzen
(auch bei Verwendung von Front-End Frameworks, z.B. Bootstrap)

⇒ Für neuen CSS Code das bequemere Cassius einsetzen

Nach Interpolation ist mischen problemlos möglich.

Julius

Julius akzeptiert gewöhnliches JavaScript, plus

- `#{}` Variablen Interpolation
- `@{}` URL Interpolation
- `^{}` Template Embedding von anderen JavaScript Templates

Sonst ändert sich nichts, auch nicht an Einrückungen!

Quasiquoter für Varianten wie CoffeScript sind auch verfügbar

Scaffolding minimiert JavaScript vor Auslieferung `hjsmin`

⇒ Es gibt inzwischen auch mehrere Ansätze, JavaScript direkt aus Haskell zu generieren, siehe z.B. Haste oder GhcJs.

Widgets

Widgets fassen einzelne Templates von verschiedenen Shakespeare-Sprachen zu einer Einheit zusammen:

```
getRootR = defaultLayout $ do
  setTitle "My Page Title"
  toWidget [lucius| h1 { color: green; } |]
  addScriptRemote "https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/j
  toWidget [julius|
    $(function() {
      $("h1").click(function(){ alert("Clicked the heading!"); });
    });
  |]
  toWidgetHead [hamlet| <meta name=keywords content="keywords">|]
  toWidget [hamlet| <h1>Here's one way for including content |]
  [whamlet| <h2>Here's another |]
  toWidgetBody [julius| alert("This is included in the body"); |]
```

Widget-Monade erlaubt kombinieren dieser Bausteine; alles wird automatisch dahin sortiert, wo es hingehört.

Widgets – whamlet

Template embedding erlaubt normalerweise nur die Einbettung aus der gleichen Template Sprache. Dagegen erlauben `whamlet` bzw. `.whamlet`-Dateien die Einbettung von Widgets in Hamlet:

```
page = [whamlet|
    <p>This is my page. I hope you enjoyed it.
    ^{footer}
  |]

footer = do
    toWidget [lucius| footer { font-weight: bold;
                               text-align: center } |]
    toWidget [hamlet|
        <footer>
          <p>That's all folks!
      |]
```

Frische Namen für frische Widgets

Bei der Kombination von Widgets könnten Namenskonflikte auftreten. Dies wird durch dynamische IDs verhindert werden:

```
getRootR = defaultLayout $ do
  headerClass <- lift newIdent
  toWidget [hamlet|<h1 .#{headerClass}>My Header|]
  toWidget [lucius| .#{headerClass} { color: green; } |]
```

Die Funktion `newIdent` erlaubt die Erzeugung von frischen IDs.

Wir müssen `lift` einsetzen, da `newIdent` nicht in der `Widget`-Monade, sondern in der `Handler`-Monade ausgeführt wird, welche ineinander verschachtelt sind.

Fortgeschrittene Funktionale Programmierung

Teil 8: Webapplikationen mit Yesod

Steffen Jost

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

11. Juli 2017

Vektoren mit statischer Grösse

Wir haben bisher Typ-Familien gesehen, welche mit einer Typklasse assoziiert waren. Das Standardbeispiel für eigenständige Typ-Familien ist durch Vektoren motiviert:

```
{-# 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: quasi Listen mit statisch bekannter Länge   Vgl. U10-1

Vektoren mit statischer Grösse

Wir haben bisher Typ-Familien gesehen, welche mit einer Typklasse assoziiert waren. Das Standardbeispiel für eigenständige Typ-Familien ist durch Vektoren motiviert:

```
{-# 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  :: * -> * -> *
```

Vektoren: quasi Listen mit statisch bekannter Länge Vgl. U10-1

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

Hello Yesod

```
{-# LANGUAGE
  OverloadedStrings,
  TypeFamilies,
  TemplateHaskell,
  QuasiQuotes
#-}

module Main where

import Yesod


data MyApp = MkApp
instance Yesod MyApp
```

```
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
|]

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
  setTitle "HelloWorld"
  toWidget [whamlet|
      <h2>Hello Yesod!
      Some text that
      is <i>displayed</i> here.
    |]

main :: IO ()
main = warp 3000 MkApp
```

Yesod Typklasse

Einstellungen einer Yesod Applikation werden mit Hilfe eines **Foundation** Datentyps vorgenommen.

Der Wert muss dazu nichts direkt speichern; Einstellungen werden über die Instanzdeklaration zur `Yesod`-Typklassen vorgenommen.

Diese Klasse fasst alle möglichen Einstellungen zusammen:

- Rendern und parsen von URLs
- Funktion `defaultLayout`
- Authentifizierung
- Sitzungsdauer
- Cookies
- Fehlerbehandlung und Aussehen der Fehlerseiten
- Externen CSS, Skripte und statische Dateien
- ...

Beispiel: eigenes Error-Handling

Für alle Einstellungen gibt es vordefinierte Standards, welche bei Bedarf überschrieben werden können:

```
data MyWebApp = MkWebApp      -- Foundation Type

instance Yesod MyWebApp       -- Defaults
```

Beispiel: eigenes Error-Handling

Für alle Einstellungen gibt es vordefinierte Standards,
welche bei Bedarf überschrieben werden können:

```
data MyWebApp = MkWebApp     -- Foundation Type

instance Yesod MyWebApp where
  errorHandler NotFound = myNotFoundHandler
  errorHandler other    = defaultErrorHandler other
```

- Default-Definition `errorHandler = defaultErrorHandler` jetzt geändert für den Fall `NotFound`.
- `myNotFoundHandler` ist selbst geschriebener Handler
- Überschreiben von `defaultLayout` verändert bereits auch Aussehen von `defaultErrorHandler`

Beispiel: Parameter

Der Foundation Type kann auch Parameter tragen. Innerhalb der **Handler** Monade kann man diese `getYesod` wieder auslesen:

```
data MyWebApp = MkWebApp { somePar  :: Int
                         , otherPar :: TVar String }

myHandler :: Handler Html
myHandler = do
  master <- getYesod                    -- master :: MyWebApp
  let myint = somePar  master           -- myint  :: Int
  mystr <- readTVarIO otherPar master -- mystr  :: String
```

- Sammelt alle statischen Parameter an einer Stellen
- `MVar`/`TVar` ermöglichen gemeinsamen Zustand für Handler

kein `IORef`

Routing & Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                   RootR     GET
  /blog/help          BlogHelpR GET
  /blog/#Int          BlogPostR GET POST
  /wiki/*WikiPfad     WikiR
  /static             StaticR   Static getStatic
|]
```

Definiert die **gesamte Sitemap** der Webapplikation.
Ausnahme: Kombination mit Yesod Unter-Websites   hier: `Static`

DSL wird innerhalb des Quasiquoters `parseRoutes` angegeben, oder in einer separaten Datei

Beim Spleißen können View-Patterns entstehen, weshalb Spracherweiterung `ViewPatterns` aktiviert sein muss

Routing

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  /blog/help         BlogHelpR GET
  /blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Zuerst wird der Pfad angegeben. Es gibt drei Arten von Pfaden:

1. Statische Pfade, z.B. `/blog/help`
2. Dynamische Pfade *enthalten* (mehrere) `/#<Typ>` Fragmente, wobei `<Typ>` eine Instanz der Klasse `PathPiece` sein muss
3. Dynamische Multipfade *enden* mit `/*<Typ>`, wobei `<Typ>` eine Instanz der Klasse `PathMultiPiece` sein muss

   Es darf auch `/+<Typ>` geschrieben werden, z.B. wegen CPP

PathPiece & PathMultiPiece

Klassen aus Modul `Yesod.Dispatch` legen lediglich Parser fest:

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece   :: s -> Text

class PathMultiPiece s where
  fromPathMultiPiece :: [Text] -> Maybe s
  toPathMultiPiece   :: s -> [Text]
```

`PathPiece`-Instanzen für `Int`, `Integer`, `String`, `Text` und `PathMultiPiece`-Instanzen für `[String]` und `[Text]` vordefiniert

Fallstrick: `read` kann Ausnahme werfen! Meist reicht schon

```
maybeRead :: Read a => String -> Maybe a
maybeRead (reads -> [(x,"")]) = Just x
maybeRead                 _   = Nothing
```

siehe auch `readMay` aus Classy-Prelude

PathPiece & PathMultiPiece

Klassen aus Modul `Yesod.Dispatch` legen lediglich Parser fest:

```
class PathPiece s where
  fromPathPiece :: Text -> Maybe s
  toPathPiece   :: s -> Text

class PathMultiPiece s where
  fromPathMultiPiece :: [Text] -> Maybe s
  toPathMultiPiece   :: s -> [Text]
```

`PathPiece`-Instanzen für `Int`, `Integer`, `String`, `Text` und `PathMultiPiece`-Instanzen für `[String]` und `[Text]` vordefiniert

Fallstrick: `read` kann Ausnahme werfen! Meist reicht schon

```
maybeRead :: Read a => String -> Maybe a
maybeRead s | [(x,"")] <- reads s = Just x
            | otherwise           = Nothing
```

siehe auch `readMay` aus Classy-Prelude

Routing

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                   RootR     GET
  /blog/help          BlogHelpR GET
  /blog/#Int          BlogPostR GET POST
  /wiki/*WikiPfad     WikiR
  /static             StaticR   Static getStatic
|]
```

Pfade werden automatisch standardisiert, gereinigt und zerlegt.

Durch Überschreiben von Default-Funktionen der `Yesod`-Typklasse kann man dies aber auch anpassen:

- `joinPath` fügt Pfadteile wieder zusammen
- `cleanPath` kümmert sich um doppelte `/`, usw.

Überlappende Routen

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  !/blog/help        BlogHelpR GET
  !/blog/#Int        BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

- Yesod kann nicht inferieren, dass `/blog/help` und `/blog/#Int` nicht überlappen
- Überlappende Routen wie etwa `/blog/#Int` und `/blog/#Text` erzeugen eine Fehlermeldung
  Dies kann mit `!` am Anfang verhindert werden
- Von oben-nach-unten erst geparste Route wird eingeschlagen
  sonst 404 Page Not Found, konfigurierbar über Foundation

Datentyp für Route

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR      GET
  !/blog/help        BlogHelpR  GET
  !/blog/#Int        BlogPostR  GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR    Static getStatic
|]
```

Der zweite Teil definiert den *Konstruktor* für die typsichere URL
Deklaration für Datentyp `Route App` wird erzeugt

erzeugter Code mit `-ddump-splices` ansehbar

- Kann direkt in URL Interpolation `@{ }` verwendet werden
  Argumente gemäß dynamischen Pfad, z.B. `@{BlogPostR 7}`
- Endung mit "R" für "Resource" ist keine zwingende Konvention

  Großbuchstabe am Anfang für Konstruktor aber schon

Datentyp für Route

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  !/blog/help        BlogHelpR GET
  !/blog/#Int        BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Der zweite Teil definiert den *Konstruktor* für die typsichere URL Deklaration für Datentyp `Route <FoundationType>` erzeugt

erzeugter Code mit `-ddump-splices` ansehbar

- Kann direkt in URL Interpolation `@{ }` verwendet werden Argumente gemäß dynamischen Pfad, z.B. `@{BlogPostR 7}`
- Endung mit "R" für "Resource" ist keine zwingende Konvention

  Großbuchstabe am Anfang für Konstruktor aber schon

Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                  RootR     GET
  !/blog/help        BlogHelpR GET
  !/blog/#Int        BlogPostR GET POST
  /wiki/*WikiPfad    WikiR
  /static            StaticR   Static getStatic
|]
```

Dritter Teil spezifiziert entweder:

1. Erlaubte HTTP-Anfragen GET,PUT,POST,DELETE,. . .
2. Keine Angabe, d.h. alle HTTP-Anfragen erlaubt
   werden dann über einen gemeinsamen Handler abgewickelt
3. Foundation Typ einer Yesod-Subsite, welche die Anfrage beantwortet; und eine Funktion, welche Wert des aktuellen Foundation Typs in den Wert des Foundation Typs der Subsite umrechnen kann. Meistens nur eine Record-Projektion

Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                   RootR     GET
  !/blog/help         BlogHelpR GET
  !/blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad     WikiR
  /static             StaticR   Static getStatic
|]
```

Für alle Anfragen muss ein **Handler** definiert werden, wobei der Name `<AnfrageTyp>` ++ `<Resource>` sein muss:

```
getRootR      :: Handler Html
getBlogHelpR  :: Handler Html
getBlogPostR  :: Int -> Handler Html
postBlogPostR :: Int -> Handler Html
handleWikiR   :: [WikiPfad] -> Handler Html
```

oder `handle` ++ `<Resource>`, falls alle HTTP-Anfragen erlaubt.

Subsite Handling

Yesod nutzt DSL zur Spezifizierung von Routen und Handling:

```
[parseRoutes|
  /                   RootR     GET
  !/blog/help         BlogHelpR GET
  !/blog/#Int         BlogPostR GET POST
  /wiki/*WikiPfad     WikiR
  /static             StaticR   Static getStatic
|]
```

- Route `/static` wird durch eine **Subsite** geleitet, welche darunterliegende Routen und Handler selbst definiert
- Im Beispiel ist `Static` der Foundation-Type der Subsite
- Angegebene Funktion `getStatic` muss Wert des Typs `Static` aus aktuellem Foundation-Type generieren können
- Scaffolding mit `yesod init` definiert Subsite für statische Resourcen um besseres Caching zu ermöglichen

Handler Monade

Handler-Funktionen leben in der Handler-Monade: `Handler Html`

```
type Handler a = HandlerT App IO a
data HandlerT site m a
```

`Handler` ist ein Typsynonym für einen Datentyp mit

`site`	Foundation Typ	
`m`	Monaden Stack	`instance MonadIO m`
`a`	Rückgabewert der Monade	

Anstellte von `Html` kann ein Handler auch Werte anderer Typen liefern, zum Beispiel eine Grafikdatei, CSS, JSON, etc.
Der content type muss jedoch bekannt sein, d.h. Handler müssen als Ergebnistyp eine Instanz von `ToTypedContent` liefern

Handling nach Content

Es ist auch möglich, Unterschiedliche Repräsentation je nach MIME-typ über eine URL abzuwickeln:

```
mkYesod "App" [parseRoutes|
  /person PersonR GET
|]

getPersonR :: Handler TypedContent
getPersonR = selectRep $ do
  provideRep $ return [shamlet| <p>Name #{name}, Age #{age} |]
  provideRep $ return $ object [ "name" .= name, "age" .= age ]
 where
  name = "Steffen" :: Text
  age  = 40 :: Int
```

Für Anfrage `/person.html` wird HTML ausgeliefert und für Anfrage `/person.json` hier das Gleiche in JSON. `Data.Aeson`

Handler Monade

Wichtige Funktionen der `Handler`-Monade:

`getYesod`	Wert des Foundation Typ, z.B. Parameter auslesen
`getUrlRender`	Renderer für Werte des `Route`-Typs
`getRequest`	Anfrage im Roh-Format
`liftIO`	zum Ausführen von IO-Aktionen
`sendFile`	Datei versenden
`setHeader`	Antwort-Header festlegen
`redirect`	Umleitung zu anderer Ressource

`notFound`, `permissionDenied` für explizite Fehlermeldung

`setCookie`, `lookupCookie` Cookies bearbeiten

`Handler` ist auch Instanz von `MonadLogger` für Logging.
Erzeugen von Log-Messages innerhalb von Templates mit
`$logError`, `$logWarn`, `$logInfo`, `$logDebug`

Webformulare

Webformulare erlauben dem Benutzer, Daten zu übermitteln.

Dies erfordert:

- Darstellung der Formulare in HTML
- Zuordnung der übermittelten Daten
- Konvertierung UTF8-Strings zu Haskell Datentypen
- Prüfungen, ob Daten im erlaubten Bereich sind
- JavaScript zu Daten-Prüfung und Bearbeitung auf dem Client
- Kombination verschiedener Formulare (-Teile)

Paket `yesod-form` stellt dies für uns bereit

Webformular Varianten in Yesod:

Applikativ Einfach zu Programmieren,
aber Kontrolle über Aussehen ist eingeschränkt

Monadisch Flexibel gestaltbare Formulare,
Verwendung jedoch etwas komplizierter

Input Spezialfall ohne HTML-Darstellung; hauptsächlich
zur Verwendung mit bestehenden Formularen

Es ist möglich, applikative Formulare automatisch in monadische umzuwandeln, und manchmal auch umgekehrt.

siehe `aformToForm` und `formToAForm`

Konvention in Yesod: Präfix je nach Variante `a`, `m` oder `i`
Z.B. Pflichtfeld in applikativen Formular erhält man mit `areq`,
ein optionales Feld in einem monadischen Formular mit `mopt`.

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder,
  `aopt` für optionale Felder eines `Maybe`-Typen
- *1. Argument* definiert Typ durch Instanz der Klasse `Field` und erklärt damit dessen Parser; für viele Typen vordefiniert
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder
  `ao`
- *1.*
  un

> Für `FieldSettings` ist eine Instanz zur Klasse `IsString` definiert, d.h. dank GHC-Erweiterung `OverloadedStrings` können solche Werte aus String-Konstanten generiert werden.

- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikatives Formular darstellen

```
carForm :: Html -> MForm Handler (FormResult Car, Widget)
carForm = renderTable carAForm

getCarR :: Handler Html
getCarR = do (widget, enctype) <- generateFormPost carForm
             defaultLayout [whamlet|
    <h2>Form Demo
    <form method=post action=@{CarR} enctype=#{enctype}>
      ^{widget}
      <button>Submit
  |]
```

- Umwandern von `AForm` in `MForm` mit `renderTable`, `renderDiv`, oder `renderBootstrap` – entscheidet über Layout des Formulars
- Erzeugen des Formulars mit `generateFormGet` oder `generateFormPost`
- `form`-Tag und Knopf zum Absenden noch nicht enthalten, damit Formulare kombiniert werden können

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
  ((result,widget), enctype) <- runFormPost carForm
    case result of
      FormSuccess car -> defaultLayout [whamlet|
              <h2>Car received:
              <p>#{show car}  |]
      _ -> defaultLayout [whamlet|
          <h2>Fehler! Nochmal eingeben:
          <form method=post action=@{CarR} enctype=#{enctype}>
          ^{widget}
          <button>Abschicken
        |]
```

Ausführen des Formulars mit `runFormGet` oder `runFormPost`

weitere Varianten möglich, z.B. `runFormPostNoNonce`

Applikatives Formular auswerten

```
postCarR :: Handler Html
postCarR = do
  ((result,widget), enctype) <- runFormPost carForm
    case result of
      FormSuccess car -> defaultLayout [whamlet|
              <h2>Car received:
              <p>#{show car}  |]
      _ -> defaultLayout [whamlet|
          <h2>Fehler! Nochmal eingeben:
          <form method=post action=@{CarR} enctype=#{enctype}>
          ^{widget}
          <button>Abschicken
        |]
```

`result` hat 3 Möglichkeiten:

- `FormSuccess a` erfolgreicher Wert
- `FormFailure Text` Parsen fehlgeschlagen
- `FormMissing` keine Daten vorhanden

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder,
  `aopt` für optionale Felder eines `Maybe`-Typen
- *1. Argument* definiert Typ durch Instanz der Klasse `Field` und erklärt damit dessen Parser; für viele Typen vordefiniert
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Applikative Formularfelder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt textField "Color" Nothing
```

Wir erklären, wie ein Wert des Typs `Car` erstellt wird:

- `areq` für erforderliche Felder
  ao
- *1.*
  un
- *2. Argument:* Bezeichner, Tooltip, id- und name-Attribut gegeben durch eine Instanz der Klasse `FieldSettings`
- *3. Argument:* Optionaler Default Wert gegeben als `Maybe`

Für `FieldSettings` ist eine Instanz zur Klasse `IsString` definiert, d.h. dank GHC-Erweiterung `OverloadedStrings` können solche Werte aus String-Konstanten generiert werden.

Defaults für Applikative Felder

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show
carAForm :: Maybe Car -> AForm Handler Car
carAForm = Car
    <$> areq textField "Model" (carModel <$> mcar)
    <*> areq intField  "Year"  (carYear  <$> mcar)
    <*> aopt textField "Color" (carColor <$> mcar)
```

Es ist oft sinnvoll ein, alle Standardwerte als kompletten Wert des Datentyps zu übergeben

Aus diesem Grund wird in Kauf genommen, dass ansonsten optionale Standardvorgaben in ein doppeltes `Maybe` verpackt werden müssen:

```
<*> aopt textField "Color" (Just $ Just "Black")
```

Spezialisierte Eingabeprüfung

```
data Car = Car { carModel :: Text, carYear  :: Int
                 carColor :: Maybe Text
               } deriving Show

carAForm :: Maybe Car -> AForm Handler Car
carAForm = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear  <$> mcar)
    <*> aopt textField    "Color" (carColor <$> mcar)
  where
    errorMessage :: Text
    errorMessage = "Your car is too old, get a new one!"

    carYearField = check validateYear intField

    validateYear y
        | y < 1990   = Left errorMessage
        | otherwise = Right y
```

Spezialisierte Eingabeprüfung

Prüfung der Eingabe erfolgt durch modifizierte Feldtypen.
Modul `Yesod.Form.Functions` bietet dazu viele Hilfsfunktionen:

```
check     :: (a -> Either msg a)      -> Field m a -> Field m aSource
checkBool :: (a -> Bool) -> msg       -> Field m a -> Field m aSource
checkM    :: (a -> m (Either msg a)) -> Field m a -> Field m aSource
```

Mit `checkBool` hätten wie beispielsweise schreiben können:

```
carYearField = checkBool (>= 1990) errorMessage intField
```

Prüfungen unter Verwendung der IO-Monade sind ebenfalls möglich mit `checkM`, z.B. um das aktuelle Datum zu ermitteln und zu prüfen, ob das eingegebene Datum in der Zukunft liegt

Eingabeprüfung mit IO

```
carYearField = checkM inPast $ checkBool (>= 1990) errorMessage

inPast y = do
    thisYear <- liftIO getCurrentYear
    return $ if y <= thisYear
        then Right y
        else Left ("You have a time machine!" :: Text)

getCurrentYear :: IO Int
getCurrentYear = do
    now <- getCurrentTime
    let today = utctDay now
    let (year, _, _) = toGregorian today
    return $ fromInteger year
```

Fallstricke Eingabeprüfung

Aufgrund der generellen Unterstützung von Yesod für Internationalisierung (Abk. "i18n") werden zur erfolgreichen Kompilation oft zusätzliche Angaben gebraucht:

- `errorMessage :: Text` explizite Typangabe oft erforderlich, da die Fehlermeldung ja generell Sprachabhängig ist

- Ebenso ist folgende Instanz-Deklaration notwendig:

  ```
  instance RenderMessage App FormMessage where
      renderMessage _ _ = defaultFormMessage
  ```

  Diese wird vom Gerüst-Tool automatisch erstellt und kann dann angepasst werden, falls Internationalisierung gewünscht wird.

Auswahllisten

```
data Car = Car { carModel :: Text, carYear :: Int
                 carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField     "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue),
              ("Grau", Gray), ("Schwarz", Black)]
```

Funktion `selectFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und kreiert eine Auswahlliste.

Auswahllisten

```
data Car = Car { carModel :: Text, carYear :: Int
                 carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq, Bounded, Enum)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [(pack $ show x,x) | x <- [minBound..maxBound]]
```

Funktion `selectFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und kreiert eine Auswahlliste.

Auswahlknöpfe

```
data Car = Car { carModel :: Text, carYear :: Int
               , carColor :: Maybe Color } deriving Show

data Color = Red | Blue | Gray | Black
    deriving (Show, Eq, Bounded, Enum)

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> aopt (radioFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [(pack $ show x,x) | x <- [minBound..maxBound]]
```

Funktion `radioFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren; also genau wie `selectFieldList` auch!

JavaScript im Formular

```
import Yesod.Form.Jquery
import Data.Time (Day)

data Car = Car { carModel :: Text, carYear :: Int
                 carRegistration :: Day
  } deriving Show

instance YesodJquery FoundT
 -- Default: jQuery Libraries at Google CDN

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
    <$> areq textField    "Model" (carModel <$> mcar)
    <*> areq carYearField "Year"  (carYear <$> mcar)
    <*> areq (jqueryDayField def
        { jdsChangeYear = True -- give a year dropdown
        , jdsYearRange = "2012:-20"
        }) "Zulassung" Nothing
```

Input Formulare ohne Layout

```
data Person = Person { personName :: Text, personAge  :: Int } deriving Show

getHomeR :: Handler Html
getHomeR = defaultLayout
    [whamlet|
        <form action=@{InputR}>
            <p>
                My name is
                <input type=text name=name>
                and I am
                <input type=text name=age>
                years old.
                <input type=submit value="Introduce myself">
    |]

getInputR :: Handler Html
getInputR = do
    person <- runInputGet $ Person
                <$> ireq textField "name"
                <*> ireq intField "age"
    defaultLayout [whamlet|<p>#{show person}|]
```

Übereinstimmung der Name-Tags muss gewährleistet werden!

Input Forms

Input Formulare

- Übereinstimmung der Name-Tags muss gewährleistet werden! Insbesondere bei dynamisch generierten Formularen problematisch.
- `ireq`/`iopt` haben nur noch zwei Argumente: Feld-Typ und Feld-Name
- Wenn die übermittelten Daten nicht passen erfolgt eine Umleitung auf eine "Invalid Arguments" Fehlerseite

Monadische Formulare

- erlauben ebenfalls eigenes Layout
- kümmern sich für uns jedoch um einzigartige Name-Tags, usw.
- `mreq`/`mopt` funktionieren wie `areq`/`aopt`, die Namen der Eingabefelder werden jedoch ignoriert (das Layout wird ja explizit angegeben)

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare

Alle Felder des Formulars werden mit monadischen Funktionen `mreq`/`mopt` beschrieben:

```
do
  (nameRes, nameView) <- mreq textField "not used" Nothing
  (ageRes , ageView)  <- mreq intField  "not used" Nothing
```

Für jedes Feld erhalten wir 2 Rückgabewerte:

1. Ergebnis des Feldes `FormResult a`, um später das Gesamtergebnis zu bauen:
   `let personRes = Person <$> nameRes <*> ageRes`
2. `FieldView` des Feldes zur Anzeige, zum Einbau in Widgets:
   `^{fvInput   ageView}` — Input-Feld
   `##{fvId     ageView}` — Id des Feldes

FieldView und FieldSettings

```
(nameRes, nameView) <- mreq textField "not used" Nothing
```

Wert des Typs `FieldView` ist Record mit den Feldern `fvLabel`, `fvTooltip`, `fvId`, `fvInput`, `fvErrors`, `fvRequired`.

Wird primär aus den Vorschlägen des zweiten Argument von `mreq`/`mopt` erzeugt, welches vom Typ `FieldSettings` sein muss.

Werte von `FieldSettings` können aus Strings erzeugt werden:

```
fromString "not used" == FieldSettings
  { fsLabel   = "not used" -- für Applikative Formulare
  , fsTooltip = Nothing
  , fsId      = Nothing
  , fsName    = Nothing
  , fsAttrs   = []
  }
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                        width: 3em;
                                        }
                              |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                              m
```

Felder werden durch zwei Teile beschrieben:

1. FormResult
2. FormView

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                        m
```

Felder werden durch zwei Teile beschrieben:

1. FormResult
2. FormView

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                          width: 3em;
                                          }
                                  |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                          ^{fvInput ageView} years old. #
                          <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHome                                                              m
```

Felder werden durch zwei Teile beschrieben:

1. FormResult
2. FormView

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                  width: 3em;
                                  }
                             |]
                    [whamlet|
                        #{extra}
                        <p> Hello, my name is ^{fvInput nameView} and I am #
                            ^{fvInput ageView} years old. #
                            <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                      width: 3em;
                                      }
                                 |]
                    [whamlet|
                       #{extra}
                       <p> Hello, my name is ^{fvInput nameView} and I am #
                           ^{fvInput ageView} years old. #
                           <input type=submit value="Introduce myself">
                    |]
    return (personRes, widget)

getHomeR = do ((res, widget), enctype) <- runFormGet personForm
              defaultLayout [whamlet|
                  <p>Result: #{show res}
                  <form enctype=#{enctype}>
                      ^{widget} |]
```

Monadische Formulare für eigenes Layout

```
data Person = Person { personName :: Text, personAge :: Int } deriving Show

personForm :: Html -> MForm Handler (FormResult Person, Widget)
personForm extra = do
    (nameRes, nameView) <- mreq textField "this is not used" Nothing
    (ageRes , ageView)  <- mreq intField  "neither is this"  Nothing
    let personRes = Person <$> nameRes <*> ageRes
    let widget = do toWidget [lucius| ##{fvId ageView} {
                                        width: 3em;
                                        }
                                |]
                  [whamlet|
                     #{extra}
                      <p> Hello, my name is ^{fvInput nameView} and I am #
                          ^{fvInput ageView} years old. #
                          <input type=submit value="Introduce myself">
```

Das extra Argument muss irgendwo ins Formular eingebaut werden.
Bei GET Formularen signalisiert es, dass das Formular abgesendet wurde.
Bei POST Formularen dient es zur Verhinderung von Cross-Site-Request-Forgery Angriffen.

```
                <form enctype=#{enctype}>
                    ^{widget} |]
```

Sessions

HTTP kennt wie Haskell keinen Zustand z.B. gut für Caching

Webapplikation benötigen aber Zustand: Login, ShoppingCart, etc.

Eine Lösung dazu sind **Sitzungen/Sessions**, d.h. eine kleine Menge an Daten (z.B. Sitzung-ID) welche der Browser *mit jeder Anfrage* an den Webserver übermittelt.

- Benutzerdaten mit Sitzungen zu speichern skaliert gut mit mehreren Servern, da jeder Request in sich abgeschlossen ist und keine zentrale Koordination/Datenbank benötigt wird
- Sitzungsdaten sollten möglichst klein sein Datenbankschlüssel
- Statischer Content sollte von separater Domain kommen, um unnötige Übertragungen von Sitzungsdaten zu verhindern

  ⇒ `static` routing

Session Control

Yesod wendet per Default automatisch *Verschlüsselung* und *Signatur* von Sitzungsdaten an, um Manipulationen zu verhindern. Sitzung verfallen automatisch nach 2 Stunden. Dies wird alles in der Yesod-Instanz des Foundation Typs eingestellt:

```
instance Yesod App where
  makeSessionBackend _ = Just <$>
      defaultClientSessionBackend minutes file
    where minutes = 2 * 60
          file    = "client-session-key.aes"
  -- Sitzungen komplett deaktivieren:
  -- makeSessionBackend _ = return Nothing
```

- Schlüssel in separater Datei gespeichert   ggf. automatisch gen.
- Ablaufdatum der Sitzung wird mit jedem Request erneuert. Yesod ignoriert abgelaufene Sitzungen automatisch.
- IP-Adresse einer Sitzung wird überprüft   ggf. abschaltbar

Session Operationen

Sitzung ist eine *ungetypte* Map:

```
type SessionMap = Map Text ByteString
```

```
getSession    :: MonadHandler m => m SessionMap
```

Liefert gesamte Sitzungs-Map – inkl. Yesod-internes

```
lookupSession :: MonadHandler m => Text -> m (Maybe Text)
```

Schlägt Schlüssel nach

```
setSession    :: MonadHandler m => Text -> Text -> m ()
```

Setzt ein Schlüssel-Wert Paar

```
deleteSession :: MonadHandler m => Text -> m ()
```

Löscht einen Schlüssel

```
clearSession  :: MonadHandler m => m ()
```

Löscht die gesamte Sitzungs-Map

Beispiel: Sessions

```
getHomeR :: Handler Html
getHomeR = do
    sess <- getSession
    defaultLayout [whamlet|
                    <form method=post>
                        <input type=text name=key>
                        <input type=text name=val>
                        <input type=submit>
                    <h1>#{show sess}
                |]

postHomeR :: Handler ()
postHomeR = do
    (key, mval) <- runInputPost $ (,) <$> ireq textField "key"
                                      <*> iopt textField "val"
    case mval of Nothing  -> deleteSession key
                 Just val -> setSession key val
    liftIO $ print (key, mval) --debug to konsole
    redirect HomeR
```

Messages

Post/Redirect/Get-Problem: Z.B. nach erfolgreichem Ausfüllen eines Formulars den Benutzer umleiten und gleichzeitig informieren, dass die Daten korrekt empfangen wurden.

Lösung jede Seite prüft Existenz eines speziellen Sitzungs-Feldes für Nachrichten. Yesod bietet eigene Schnittstelle dafür:

```
setMessage ::  MonadHandler m => Html -> m ()
```
Setzt eine Message in der Sitzung.

```
getMessage :: MonadHandler m => m (Maybe Html)
```
Liest letzte Message aus und löscht diese, sofern vorhanden.

`defaultLayout` nutzt `getMessage` um ggf. Botschaft anzuzeigen
⇒ bei Überschreiben von `defaultLayout` die Behandlung von `getMessage` nicht vergessen!

Beispiel: Messages

```
getA = do
  page <- defaultLayout $ do
    [whamlet|
      You are at A
    |]
    links
  setMessage "Previous: A"
  return page

getB = do
  maybemsg <- getMessage
  page <- defaultLayout $ do
    [whamlet|
      <p>You are at B
      $maybe msg <- maybemsg
        <p>Message: #{msg}
    |]
    links
  setMessage "Previous: B"
  return page
```

Ultimate Destination

Erlaubt temporäre Umleitung eines Benutzers (z.B. für Login); danach wird Benutzer wieder zur ursprünglichen Seite geschickt

```
setUltDest ::(MonadHandler m, RedirectUrl (HandlerSite m) url) =>
             url -> m ()
```
Setzt Ultimate Destination

```
setUltDestCurrent :: MonadHandler m => m ()
```
Setzt Ultimate Destination auf aktuelle Seite, falls $\neq$ 404

```
setUltDestReferer :: MonadHandler m => m ()
```
Setzt Ultimate Destination auf Referer = vorherige Seite

```
redirectUltDest   :: ... =>  url -> m a
```
Führt redirect auf Ultimate Destination aus und löscht diese, falls gesetzt, sonst redirect auf angegebene URL

```
clearUltDest      :: MonadHandler m => m ()
```
Löscht Ultimate Destination.

Beispiel: Ultimate Destination

```
getSayHelloR = do -- Display name or request it
    mname <- lookupSession "name"
    case mname of
        Nothing -> do
            setUltDestCurrent
            setMessage "Please tell me your name"
            redirect SetNameR
        Just name -> defaultLayout [whamlet|<p>Welcome #{name}|]

getSetNameR = defaultLayout -- Display form
    [whamlet|
        <form method=post>
            My name is #
            <input type=text name=name>
            . #
            <input type=submit value="Set name">
    |]

postSetNameR = do -- Evaluate Form & set in session
    name <- runInputPost $ ireq textField "name"
    setSession "name" name
```

Persistenz

Manchmal sollen Daten länger als eine Sitzung halten.

`Database.Persist` und `Yesod.Persistent` stellen dazu eine *typsichere* Schnittstelle für Standard-Datenbanken bereit. Typsicherheit gilt auch dann, wenn die eigentliche Datenbank selbst ungetypt ist!

- `Persistent` unterstützt verschiedene Datenbanken: SQLite, PostgreSQL, MySQL, MongoDB, ...
- `Persistent` führt viele SQL Migrationen automatisch aus

Modul `Database.Persist` ist *unabhängig von Yesod*, und kann generell zur Anbindung einer Datenbank an ein Haskell Programm verwendet werden.

Typ-Sichere Persistenz

Datenbanken werden mit TemplateHaskell spezifiziert:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
        name String
        age Int
        deriving Show
    BlogPost
        title String
        authorId PersonId
    |]
```

Definiert Hilfsfunktionen und Haskell-Datentypen:

```
data Person { personName :: String, personAge :: Int }
        deriving (Show, Read, Eq)
type PersonId = Key Person

data BlogPost { blogPostTitle    :: String,
                blogPostAuthorId :: PersonId }
        deriving (Read, Eq)
```

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persistLowerCase|
    Person
        name String
        age Int Maybe
        deriving Show
    BlogPost
        title String
        authorId PersonId
        deriving Show
    |]

main :: IO ()
main = runSqlite "dbfile.sql" $ do
    runMigration migrateAll
    johnId <- insert $ Person "John Doe" $ Just 35
    janeId <- insert $ Person "Jane Doe" Nothing

    insert $ BlogPost "My fr1st p0st" johnId
    insert $ BlogPost "One more for good measure" johnId

    oneJohnPost <- selectList [BlogPostAuthorId ==. johnId] [LimitTo 1]
    liftIO $ print (oneJohnPost :: [Entity BlogPost])
    john <- get johnId
    liftIO $ print (john :: Maybe Person)
    delete janeId
    deleteWhere [BlogPostAuthorId ==. johnId]
```

Benutzerspezifische Datenbankfelder

Feldtypen müssen Instanzen der Klasse `PersistField` sein.
Für Enumerations kann die Instanz-Deklaration mit einer TemplateHaskell Funktion automatisch abgeleitet werden:

```
data Employment = Employed | Unemployed | Retired
    deriving (Show, Read, Eq)
derivePersistField "Employment"

-- Andere Datei:
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
[persistLowerCase|
Person
    name String
    employment Employment
|]
```

Konstruktoren werden in der Datenbank als String gespeichert, welche mit `Show` und `Read` verarbeitet werden. Dies erlaubt nachträgliche Erweiterung der Konstruktoren.

Uniqueness

Zeilen, welche mit Großbuchstaben beginnen, spezifizieren Einschränkung zu Einzigartigkeit von Datenbankeinträgen:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName
      deriving Show
  |]
```

Es wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also zum Schlüssel:

```
getBy $ UniquePerson "Steffen"
```

UNIQUENESS

Zeilen, welche mit Großbuchstaben beginnen, spezifizieren Einschränkung zu Einzigartigkeit von Datenbankeinträgen:

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int
      UniquePerson firstName lastName
      deriving Show
  |]
```

Es wird ein Konstruktor generiert, der gezieltes Nachschlagen mit der Funktion `getBy` erlaubt, das Feld wird also zum Schlüssel:

```
getBy $ UniquePerson "Steffen" "Jost"
```

Werden mehrere Felder angegeben, dann muss nur die entsprechende Kombination einzigartig sein.

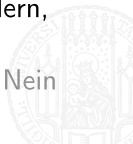
Optionale Felder

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int Maybe
      deriving Show
  |]
```

`Maybe` wird ganz *hinten* angestellt und macht das Feld optional in der Datenbank "nullable"

Achtung: Uniqueness funktioniert nur mit nicht-optionalen Feldern, da SQL nicht festlegt ob `NULL==NULL` gilt

Haskell: Ja, PostgreSQL: Nein



Nachträgliche Erweiterung

Persist kann sich auch um nachträgliche Änderungen an der Datenbank kümmern und übernimmt die **Migration**

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
      firstName String
      lastName String
      age Int Maybe
      timestamp UTCTime default=CURRENT_TIME
      deriving Show
|]
```

- Hinzufügen optionaler Felder ist problemlos   auf NULL gesetzt
- Mit `default` können Standardwerte vorgegeben werden
  Achtung: Dies ist Datenbank spezifisch, z.B. `CURRENT_TIME` ist hier eine spezielle Funktion der Datenbank

Datenbank Migration

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
  [persistLowerCase|
    Person
        name String
        age Int
  |]
```

Automatische Migration mit Funktion `runMigration` bei:

- Zusätzliche Datentypen hinzufügen
- Zusätzliche Felder mit Default hinzufügen
- Typwechsel bei Felder, sofern Konversion möglich
  z.B. Wechsel zwischen Optional- und Pflichtfeld

Manuelle Intervention notwendig bei: `runMigrationUnsafe`

- Felder löschen
- Umbenennung von Felder oder Datentypen

Aktionen werden auf `stderr` protokolliert;
Migrations-Vorschau mit `printMigration` möglich

Datenbank Schnittstelle

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll -- ggf. Migration durchführen
  steffenId <- insert $ Person "Steffen" 37 -- Einfügen
  steffen   <- get steffenId                -- Abfragen
  liftIO $ print steffen
```

`runSqlite` Datenbank einbinden, je nach Argument:

- `":memory:"` Datenbank im Speicher
- `"myfile.sql"` Datenbank in Datei

Genau eine Datenbank Transaktion pro Aufruf von `runSqlite`
Atomar, d.h. alle Aktionen bei Ausnahme zurückgenommen

`runMigration` Default-Migration; mindestens einmal bei Erstellen einer neuen Datenbank durchführen!

Datenbank Schnittstelle

Die Klasse `PersistStore b m` definiert u.a.:

`insert  :: ... => val -> m (Key val)`
Wert in Datenbank einfügen

`get     :: ... => Key b val -> m (Maybe val)`
Schlüssel in Datenbank nachschlagen

`getBy   :: ... => Unique val -> m (Maybe (Entity val))`
Unique nachschlagen, liefert `Entity valId val`

`delete  :: ... => Key val -> m ()`
Schlüssel in Datenbank löschen

`repsert :: ... => Key val -> val -> m ()`
Schlüssel ggf. ersetzen

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll
  steffenId <- insert $ Person "Steffen" 37
  steffen <- get steffenId
  liftIO $ print steffen
```

Datenbank Abfragen

Neben `get` und `getBy` gibt es noch echte Datenbank Abfragen:

```
selectList :: ... =>
    [Filter val] -> [SelectOpt val] -> m [Entity val]
```

Zwei Argumente:

- Liste von Filtern
- List von Auswahl Optionen

Beispiel: Alle Menschen zwischen 26 und 30 auswählen:

```
people25bis30 <- selectList
  [PersonAge >. 25, PersonAge <=. 30] []
```

Datenbank Abfragen

- Liste der Filter ist UND-Verknüpft
- Operatoren wie gewohnt, nur mit Punkt am Ende
- `!=.` anstelle von `/=.` wegen Namenskonflikt
- `<-.` und `/<-.` stehen für "element" und "nicht element"

Beispiel: Alle Menschen zwischen 26 und 30, oder deren Namen nicht "Adam" oder "Bonny" lautet, oder deren Alter genau 50 oder 60 beträgt:

```
people <- selectList
  (       [PersonAge >. 25, PersonAge <=. 30]
      ||. [PersonFirstName /<-. ["Adam", "Bonny"]]
      ||. ([PersonAge ==. 50] ||. [PersonAge ==. 60])
  )
  []
```

Datenbank Abfragen

Auswahl Optionen:

- `Asc` *Feld* für aufsteigend sortierte Ergebnisse
- `Desc` *Feld* für absteigenden sortierte Ergebnisse
- `LimitTo n` um Anzahl Ergebnisse zu Begrenzen
- `OffSetBy n` um die ersten n-Ergebnisse zu überspringen

```
let resultsPerPage = 10
selectList
    [ PersonAge >=. 18 ]
    [ Desc PersonAge
    , Asc PersonLastName
    , Asc PersonFirstName
    , LimitTo resultsPerPage
    , OffsetBy $ (pageNumber - 1) * resultsPerPage
    ]
```

Datenbank Abfragen

Alternative Abfragen:

```
selectList :: ... =>
    [Filter val] -> [SelectOpt val] -> m [Entity val]
```

liefert Ergebnis-Liste

```
selectFirst :: ... =>
    [Filter val] -> [SelectOpt val] ->  m (Maybe (Entity val))
```

liefert nur das erste Ergebnis

```
selectKeys :: PersistEntity val =>
    [Filter val] -> Source (ResourceT (b m)) (Key val)
```

liefert nur die Schlüssel der Ergebnisse

Direkte, rohe, typ-unsichere SQL Operationen sind auch möglich, z.B. für Joins

Datenbank Manipulationen

```
update :: PersistEntity val =>
  Key val -> [Update val] -> m ()
```

Datenbankwert verändern

```
updateWhere :: PersistEntity val =>
  [Filter val] -> [Update val] -> m ()
```

nur spezielle Werte verändern

```
deleteWhere :: PersistEntity val =>
  [Filter val] -> m ()
```

nur spezielle Werte löschen

```
personId <- insert $ Person "Steffen" "Jost" 39
update personId [PersonAge =. 40]

updateWhere [PersonFirstName ==. "Steffen"]
            [PersonAge +=. 1]
```

Operatoren: `=.`, `+=.`, `-=.`, `*=.`, `/=.`

Datenbanken Integration in Yesod

Datenbank/Yesod-Schnittstelle in `Yesod.Persist` definiert:

`runDB :: YesodDB site a -> HandlerT site IO a`

Datenbank Zugriff in Handler Monade

`get404 :: ... => Key val -> m val`

Wie `get`, nur liefert direkt 404 bei fehlschlag `getBy404`

- `YesodPersist`-Instanz des Foundation Types hält fest, welche Datenbank verwendet wird.
- Foundation Typ erhält ein Argument für die Datenbank, damit diese überall zugänglich ist.

- Yesod Scaffolding Tool kümmert sich bereits um alles

Beispiel: Datenbank in Yesod

Minimales Yesod-Beispiel ändern/erweitern um:

```
data App = App ConnectionPool -- Parameter für Foundation

instance YesodPersist PersistTest where
    type YesodPersistBackend PersistTest = SqlBackend
    runDB action = do
        App pool <- getYesod
        runSqlPool action pool

openConnectionCount :: Int
openConnectionCount = 10

main :: IO ()
main = runStderrLoggingT $ withSqlitePool "myfile.db3"
  openConnectionCount $ \pool -> liftIO $ do
    runResourceT $ flip runSqlPool pool $ do
        runMigration migrateAll
        insert $ Person "Michael" "Snoyman" 26
    warp 3000 $ PersistTest pool

getPersonR :: PersonId -> Handler String
getPersonR personId = do  person <- runDB $ get404 personId
                          return $ show person
```

Datenbank Zugriff in Widgets

Keine Datenbankanfragen innerhalb Widgets erlaubt:

```
[whamlet|
  <ul>
    $forall Entity blogid blog <- blogs
      $with author <- runDB $ get404 $ blogAuthor -- Error
        <li>
          <a href=@{BlogR blogid}>
            #{blogTitle blog} by #{authorName author}
|]
```

Problem:

Innerhalb des Widgets befinden wir uns nicht mehr in der `Handler`-Monade!

Datenbank Zugriff in Widgets

Lösung Abfrage vorher durchführen:

```
getHomeR :: Handler Html
getHomeR = do blogs <- runDB $ selectList [] []
              defaultLayout $ do
                setTitle "Blog posts"
                [whamlet|
                  <ul>
                    $forall blogEntity <- blogs
                      ^{showBlogLink blogEntity}
                |]
showBlogLink :: Entity Blog -> Widget
showBlogLink (Entity blogid blog) = do
  author <- handlerToWidget $ runDB $ get404 $ blogAuthor blog
  [whamlet|
      <li>
          <a href=@{BlogR blogid}>
              #{blogTitle blog} by #{authorName author}
  |]
```

Weitere Themen:

Yesod

- Authentifizierung: Wer macht den Zugriff?
- Autorisierung: Wer darf welchen Zugriff machen?
- Internationalisierung:
  Eine Seite in mehreren Sprachen ausliefern
- Subsites:
  Webapp aus Bausteinen zusammensetzen

Persist

- Esqueleto: Separate Zusatz-Bibliothek
  Typsichere DSL generiert rohe SQL-Anfragen
  z.B. nützlich für Joins

Quellenangabe

Dieses Kapitel zeigte Ideen und Code-Beispiele unter anderem aus folgenden Quellen:

- Michael Snoyman. "Haskell and Yesod". O'Reilly, April 2012, ISBN 978-1-449-31697-6
- Dokumentation des Frameworks auf http://www.yesodweb.com
- Tutorials der School of Haskell auf https://www.fpcomplete.com/school
- Oleg Kiselyov, Simon Peyton Jones, Chung-chieh Shan. "Fun with Type Functions". Proceedings of Tony Hoare's 75th birthday celebration.
- Philip Wadler."Views: A way for pattern matching to cohabit with data abstraction". POPL 1987.
- Dokumentation von GHC auf https://www.haskell.org