Fortgeschrittene Funktionale Programmierung mit Haskell

ViewPatterns, GADTs, Webserver mit Yesod (Teil 2)

Steffen Jost

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

7. Januar 2016

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 `toAscList` erlaubt uns eine *Ansicht* auf den 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 #-}
getMinKeyValVP :: Map k v -> Maybe (k,v)
getMinKeyValVP (toAscList -> (h:_)) = Just h
getMinKeyValVP _                    = 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"
```

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 -> ???
```

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)
```

$\Rightarrow$ 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)

Spracherweiterung **Generalised Algebraic Datatypes (GADTs)** verallgemeinert dieses Konzept:

- Konstruktoren werden 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

Hello Yesod

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}
module Main where

import Yesod

data App = App      -- Foundation Type
instance Yesod App

-- Routing
mkYesod "App" [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 App  -- Port
```

Yesod Typklasse

Jede Yesod Applikation muss eine Instanz der `Yesod`-Klasse sein. Genauer, der *Foundation* Datentyp muss als Instanz deklariert werden. Dieser Datentyp ist simpel, kann aber parametrisiert sein.

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

Durch explizite Definition in der Instanzdeklaration kann man Defaults bei Bedarf auch überschreiben.

Beispiel: eigenes Error-Handling

Durch explizite Definition in der Instanzdeklaration kann man Defaults bei Bedarf auch überschreiben:

```
data MyWebApp = MyWebApp    -- Foundation Type

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

- Hier überschreiben wir den Default für `errorHandler` um eine spezialisierte Webseite im Fehlerfall `NotFound` anzuzeigen.
- Diese Webseite wird hier durch unsere selbst geschriebene Funktion `myNotFoundHandler` erzeugt
- Das Aussehen des vordefinierten `defaultErrorHandler` kann man aber z.B. auch durch Überschreiben von `defaultLayout` anpassen.

Beispiel: statische Parameter

Der Foundation Type darf auch Parameter tragen. innerhalb der **Handler** Monade kann man den Wert des Foundation Typs mit `getYesod` wieder auslesen:

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

myHandler :: Handler Html
myHandler = do
  master <- getYesod
  myint    = somePar  master
  mystring = otherPar master
```

Dies ist nützlich, um alle statische Parameter einer Webapplikation global an einer Stelle zu sammeln.

Routing & Handling

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
[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

Diese DSL wird innerhalb des Quasiquoters `parseRoutes` angegeben, oder aber in einer separaten Datei wie bei `yesod init`

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

Routing

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
[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:

- Statische Pfade, z.B. `/blog/help`
- Dynamische Pfade *enthalten* (mehrere) `/#<Typ>` Fragmente, wobei `<Typ>` eine Instanz der Klasse `PathPiece` sein muss
- 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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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 erste geparste Route wird eingeschlagen
  sonst 404 Page Not Found, konfigurierbar über Foundation

Datentyp für Route

Yesod nutzt eine DSL zu Spezifizierung Routen und Handlings:

```
[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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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 eine DSL zu Spezifizierung Routen und Handlings:

```
[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` speichert Foundation Typ
- `m` verschachtelte Monade
- `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` (hoffentlich) das Gleiche in JSON.

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 Resource
`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 all dies für uns bereit!

Webformular Varianten in Yesod:

Applikativ Einfach zu Programmieren,
aber Kontrolle über Aussehen ist eingeschränkt

Monadisch Flexibel gestaltbare Formulare,
jedoch etwas komplizierter zu Programmieren

Input Speziallfall 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`.

Applikatives 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`

Applikatives 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.

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

Fortgeschrittene Funktionale Programmierung mit Haskell

Webserver mit Yesod (Teil 3)

Steffen Jost

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

14. Januar 2016

Werbe-Einblendung des Lehrstuhl PMS

Tutoren für ProMo gesucht

Für die Vorlesung Programmierung und Modellierung (SoSe 2016) sucht der Lehrstuhl PMS studentische Hilfskräfte.
Hauptaufgabe ist das Betreuen der wöchentlichen Übungen.

Bei Interesse bitte melden bei:

Dr. Clemens Schefels
clemens.schefels@ifi.lmu.de
Raum E112 an der Oettingenstraße 67

Werbe-Einblendung des Lehrstuhl TCS

Unser Lehrstuhl TCS sucht für das kommende Sommersemester Tutoren/Korrektoren für folgende Veranstaltung:

- Logik & Diskrete Strukturen
- Algorithmen & Datenstrukturen

Bei Interesse bitte melden bei:

Dr. Steffen Jost
jost@tcs.ifi.lmu.de
Raum E111 an der Oettingenstraße 67

Anmeldung Projektabnahme per UniworX

Bitte tragen Sie sich *ab jetzt bis spätestens 4.2.* für einen Termin zur Abnahme Ihres des Abschlußprojektes zur Vorlesung "Fortgeschrittene Funktionale Programmierung" per UniworX ein!

Geben Sie eine Textdatei mit folgenden Information ab:

- Namen von bis zu drei Teilnehmern
  Bitte nur eine Abgabe! UniworX-Abgabe-Gruppen bilden!
- Einschränkung/Wünsche beim Prüfungstermin
  Welche Uhrzeiten/Tage gehen keinesfalls?
  Ausschluß der Öffentlichkeit gewünscht?
- Projekttitel und kurze Beschreibung des Projekts
  Welche Bezüge zur Vorlesung?

Die Prüfung finden 14.3.–18.3.2016 statt.

Formlose Abmeldungen bis 13.3. möglich

Ideen

- Abschlussprojekte des Vorjahres:
  www.tcs.ifi.lmu.de/lehre/ws-2014-15/fun/projekte/galerie
  www.tcs.ifi.lmu.de/lehre/ws-2014-15/fun/fun#projekttable
- Yesod-Webapp: UniworX-Clone, Spiel, Planer/Verwaltung, . . .
  Hinweis: Hauptaugenmerk liegt auf Server-Seite; wo ist Bezug zur Vorlesung, wenn alles wesentliche in Javascript ist?
- Spiel mit GUI, eventuell auch mit AI
- Parallele Rechenintensive Hilfstools, z.B. wie in A7-3
  per Kommandozeile, Web-Interface oder GUI
- Interpreter/Parser für eine andere Sprache / DSL
  Techniken aus Übungen zum Lambda-Kalkül verwendbar
- Kommandozeilen Tools wie im Buch “Real-World Haskell” behandelt

Applikatives 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`

Applikatives 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 genrellen 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
```

Jedes Feld erhalten wir so 2 Teile:

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
                    ^{widget} |]
```

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
                    ^{widget} |]
```

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
                    ^{widget} |]
```

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

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.

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. Sitzungs-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

$\Rightarrow$ `static` routing

Session Control

Yesod wendet per Default automatisch *Verschlüsselung* und *Signatur* von Sitzungsdaten an, um Manipulationen zu verhinden. 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
$\Rightarrow$ 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)
```

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`
Atomisch, d.h. alle Aktionen bei Ausnahme zurückgenommen

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

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`
- Philip Wadler."Views: A way for pattern matching to cohabit with data abstraction". POPL 1987.
- Dokumentation von GHC auf `https://www.haskell.org`