Funktionale Programmierung

Web Applications mit Yesod

Andreas Abel und Steffen Jost

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

25. Juni 2012

Records

```
data Person' = Mann'  String  Int
             | Frau'  String  Double

data Person  = Mann { name:: String, alter  :: Int    }
             | Frau { name:: String, gewicht:: Double }
```

Der Datenyp `Person` kann exakt genauso verwendet werden, wie `Person'`, auch mit der gleichen Syntax:

```
p1 :: Person = Mann "Steffen" 37
```

```
instance Show Person where
  show (Mann n a) = n ++ "(" ++ (show a) ++ ")"
```

Records

```
data Person' = Mann'  String  Int
             | Frau'  String  Double

data Person  = Mann { name:: String, alter  :: Int    }
             | Frau { name:: String, gewicht:: Double }
```

Der Datenyp `Person` kann exakt genauso verwendet werden, wie `Person'`, auch mit der gleichen Syntax:

```
p1 :: Person = Mann "Steffen" 37
p2 :: Person=  Frau { gewicht=99.9, name="MsPiggy" }

instance Show Person where
  show (Mann n a)  =  n ++ "(" ++ (show a) ++ ")"
  show  Frau { name=n, gewicht=g } = n
```

Alternativ können auch Feldnamen verwendet werden;
die Reihenfolge ist dann beliebig.

Records – können mehr

```
data Person = Mann { name:: String, alter :: Int }
            | Frau { name:: String, gewicht:: Double}
```

Pattern matching auf Record-Felder darf auch partiell sein:

```
mannAlter Mann {alter=x} = x

isMann Mann {} = True
isMann  _      = False
```

Es werden automatisch partielle Projektionen definiert:

```
name    :: Person -> String
alter   :: Person -> Int
gewicht :: Person -> Double
```

```
> name p1
"Steffen"
> gewicht p1
*** Exception: No match in record selector gewicht
```

Records – können mehr

```
data Person  = Mann { name:: String, alter  :: Int    }
             | Frau { name:: String, gewicht:: Double }
```

Es gibt auch funktionale "Field-updates" (Kopien werden erstellt)

```
p1  = Mann { name="Steffen", alter=37 }
p3  = p1 { name = "Andreas" }
p3' = p3 { alter= 1 + alter p3 }
```

Record-Typen kann man ohne große Code-Änderungen nachträglich erweitern, denn

```
p4  = Frau { name="Angela" }
```

ist identisch zu

```
p4  = Frau { name="Angela", gewicht=undefined }
```

GHC gibt aber entsprechende Warnungen heraus.

Data.Functor

```
class Functor f  where
  fmap  :: (a -> b) -> f a -> f b
  (<$)  ::      b   -> f a -> f b
```

Die Klasse `Functor` verallgemeinert die `map` Funktion für parameterisierte Typen.

Es sollten folgende Gesetze gelten:

Identität `fmap id = id`

Komposition `fmap (f . g) = fmap f . fmap g`

Oft wird `fmap` auch als Infix verwendet:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Jede Monade ist auch ein Funktor via `liftM`:

```
liftM (\x -> x+1) [1,2,3] = [2,3,4]
```

Control.Applicative

Applicative verallgemeinert ein Muster, welches wir von Monaden kennen. Jede Monade ist auch eine Instanz von Applicative.
Es sollte `pure` gleich `return` und <*> gleich `ap` gelten.

```
class Functor f => Applicative f where
  pure  :: a -> f a                       -- lifting
  (<*>) :: f (a -> b) -> f a -> f b       -- application
  (*>)  :: f a -> f b -> f b              -- seq-right
  (<*)  :: f a -> f b -> f a              -- seq-left
```

Identität `pure id <*> v = v`
Komposition `pure (.) <*>u <*>v <*>w = u <*>(v<*>w)`
Homomorphie `pure f <*> pure x = pure (f x)`
Interchange `u <*> pure y = pure ($ y) <*> u`

Code sieht "funktionaler" aus, wenn man anstatt do-Notation die Funktionen `ap`, `liftM` und `liftM2` verwendet.
Kommt man damit aus, ist `Applicative` die bessere Wahl.

Control.Applicative – Beispiel

```
sequence :: [IO a] -> IO [a]
sequence []  = return []
sequence (c : cs) = do
  x <- c
  xs <- sequence cs
  return (x : xs)
```

kann man im applikativen Stil auch so schreiben:

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (c : cs) = return (:) `ap` c `ap` sequence cs
```

Das Muster $\texttt{return} \cdot \texttt{ap} \cdots \texttt{ap}$ findet man häufiger und wird im Module `Applicative` durch `pure` und <*> verallgemeinert

```
sequence (c : cs) = (:) <$> c <*> sequence cs
```

Webformulare

Webformulare erlauben dem Benutzer, Daten zu übermitteln.

Dies erfordert:

- Prüfungen, ob Daten im erlaubten Bereich sind
- Marshalling zu Haskell Datentypen
- JavaScript zu Daten-Prüfung und Bearbeitung auf dem Client
- Darstellung der Formulare in HTML
- Kombination verschiedener Formulare

Das Paket `yesod-form` stellt diese alle für uns bereit!

Webformular Varianten in Yesod:

Applikativ Einfach zu Programmieren, aber Kontrolle über Formular-Aussehen ist eingeschränkt

Monadisch Flexibler als applikative Formulare, jedoch etwas komplizierter zu Programmieren

Input Speziallfall ohne HTML-Darstellung; hauptsächliche zur Verwendung mit bestehendem 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 Formularbeispiel

```
data Car = Car { carModel :: Text, carYear :: Int  }

carAForm :: AForm FoundT FoundT Car
carAForm = Car <$> areq textField "Model" Nothing
               <*> areq intField  "Jahr"  (Just 2012)

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

getCarR :: Handler RepHtml
getCarR = do
    ((result, widget), enctype) <- runFormGet carForm
    case result of
        FormSuccess car ->
          defaultLayout [whamlet|<p>#{show car}|]
        _ -> defaultLayout [whamlet|
<form method=get action=@{CarR} enctype=#{enctype}>
    <table>
        ^{widget}
    <input type=submit>
|]
```

`renderTable`/ `renderDiv` wandeln `AForm` in `MForm` um und entscheiden über Layout des applikativen Formulars

Applikatives Formularbeispiel

```
data Car = Car { carModel :: Text, carYear :: Int  }

carAForm :: AForm FoundT FoundT Car
carAForm = Car <$> areq textField "Model" Nothing
               <*> areq intField  "Jahr"  (Just 2012)

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

getCarR :: Handler RepHtml
getCarR = do
    ((result, widget), enctype) <- runFormGet carForm
    case result of
        FormSuccess car ->
          defaultLayout [whamlet|<p>#{show car}|]
        _ -> defaultLayout [whamlet|
<form method=get action=@{CarR} enctype=#{enctype}>
    <table>
        ^{widget}
    <input type=submit>
|]
```

Zum Ausführen des Formulars:

- `runFormGet`
- `runFormPost` / `runFormPostNoNonce`
- `generateFormGet` / `generateFormPost` ignoriert Parameter

Applikatives Formularbeispiel

```
data Car = Car { carModel :: Text, carYear :: Int  }

carAForm :: AForm FoundT FoundT Car
carAForm = Car <$> areq textField "Model" Nothing
               <*> areq intField  "Jahr"  (Just 2012)

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

getCarR :: Handler RepHtml
getCarR = do
    ((result, widget), enctype) <- runFormGet carForm
    case result of
        FormSuccess car ->
          defaultLayout [whamlet|<p>#{show car}|]
        _ -> defaultLayout [whamlet|
<form method=get action=@{CarR} enctype=#{enctype}>
    <table>
        ^{widget}
    <input type=submit>
|]
```

`result` hat 3 Möglchkeiten:

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

Applikative Felder

```
data Car = Car { carModel :: Text, carYear :: Int

  } deriving Show

carAForm :: AForm FoundT FoundT Car
carAForm = Car
  <$> areq textField "Model" Nothing
  <*> areq intField  "Jahr"  (Just 2012)
```

Ein Feld braucht:

Typ gegeben als Instanz der Klasse `Field`.
Vordefiniert für `Int`, `Double`, `Text`
und auch für eMail, Uhrzeit, Datum, usw.

Bezeichner gegeben Instanz der Klasse `FieldSettings`,
für Feldbezeichner, Tooltip, id und name Attribute
String ist Instanz, welche den Bezeichner festlegt

Default gegeben als Maybe

Optionale Applikative Felder

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

carAForm :: AForm FoundT FoundT Car
carAForm = Car
  <$> areq textField "Model" Nothing
  <*> areq intField  "Jahr"  (Just 2012)
  <*> aopt textField "Farbe" Nothing
```

- Optionale Felder entsprechen `Maybe`-Typen
- Default wird per `Maybe`-Typ verpackt
- Default für optionale Felder doppelt in `Maybe` verpackt

Optionale Applikative Felder

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

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
  <$> areq textField "Model" (carModel <$> mcar)
  <*> areq intField  "Jahr"  (carYear  <$> mcar)
  <*> aopt textField "Farbe" (carColor <$> mcar)
```

Es kann oft sinnvoll ein, alle Vorgaben als komplette Wert des Datentyps zu übergeben.

Dazu ist das doppelte Maybe für Default-Werte praktisch.

Eingabeprüfung

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

carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
  <$> areq textField    "Model" (carModel <$> mcar)
  <*> areq carYearField "Jahr"  (carYear  <$> mcar)
  <*> aopt textField    "Farbe" (carColor <$> mcar)
 where
    errorMessage :: Text
    errorMessage = "Das Auto ist leider zu alt!"

    carYearField = check validateYear intField

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

Eingabeprüfung

Prüfung der Eingabe erfolgt durch modifizierte Feldtypen.
Diese schriebt man nicht neu, sondern generiert Sie mit `check`:

```
check :: RenderMessage master msg => (a -> Either msg a)
                  -> Field sub master a -> Field sub master a
```

Abkürzung für Boolesche Tests:

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

Prüfungen unter Verwendung der IO-Monade sind ebenfalls möglich mit `checkM`

Eingabeprüfung mit IO

```
carAForm :: Maybe Car -> AForm FoundT FoundT Car
carAForm mcar = Car
  <$> areq textField    "Model" (carModel <$> mcar)
  <*> areq carYearField "Jahr"  (carYear  <$> mcar)
  <*> aopt textField    "Farbe" (carColor <$> mcar)
 where
    errorMessage :: Text
    errorMessage = "Das Auto ist leider zu alt!"

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

    inPast y = do
        thisYear <- liftIO getCurrentYear
        return $ if y <= thisYear
            then Right y
            else Left ("Das Auto ist zu neu!" :: Text)

getCurrentYear :: IO Int
```

Eingabeprüfung

`errorMessage :: Text` explizite Typangabe oft erforderlich, wegen Überladung zur Unterstützung von Internationalisierung.

Ebenso folgende Instanz-Deklaration notwendig:

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

Diese wird vom Gerüst-Tool automatisch erstellt und kann dann angepasst werden.

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), (
```

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
```

Monadische Formulare

Monadische Formulare sind notwendig für spezielle Layouts:

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

personForm :: Html -> MForm FoundT FoundT (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)
```

Sessions

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

Applikation mit Logins, Blogs, Twitter oder WebShops benötigen aber einen Zustand. Eine Lösung sind Sessions, welche mit jedem Request an den Webserver übermittelt werden.

- Benutzerdaten in Sitzungen zu Speichern skaliert gut, da jeder Request in sich abgeschlossen ist und keine 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

Achtung, Sitzungsdaten sind ungetypt:

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

Session Control

Yesod wendet per Default automatisch **Verschlüsselung** und **Signatur** von Sitzungsdaten an, um Manipulationen zu verhinden.

- Schlüssel in separater Datei gespeichert ggf. automatisch gen.

```
encryptKey :: Yesod a =>
              a -> IO (Maybe Web.ClientSession.Key)
```

- Sitzungsdauer wird mit der Yesod-Instanz des Foundation Typs (vor-)eingestellt

```
clientSessionDuration :: Yesod a => a -> Int
```

  Ablaufdatum der Sitzung wird mit jedem Request erneuert. Yesod ignoriert abgelaufene Sitzungen auch selbst.

- IP-Adresse einer Sitzung wird überprüft ggf. abschaltbar

Session Operationen

Sitzungs-Map: `type SessionMap = Map Text ByteString`

```
getSession :: GHandler sub master SessionMap
```
Liefert gesamte Sitzungs-Map

```
setSession :: Text -> Text -> GHandler sub master ()
```
Setzt ein Schlüssel-Wert Paar

```
deleteSession :: Text -> GHandler sub master ()
```
Löscht einen Schlüssel

```
clearSession :: GHandler sub master ()
```
Löscht die gesamte Sitzungs-Map

```
{-# LANGUAGE TypeFamilies, QuasiQuotes, TemplateHaskell, MultiParamTypeClasses, Overload
import Yesod
import Control.Applicative
import qualified Web.ClientSession as CS

data SessionExample = SessionExample
mkYesod "SessionExample" [parseRoutes|
    / Root GET POST
  |]

getRoot :: Handler RepHtml
getRoot = do
    sess <- getSession
    hamletToRepHtml [hamlet|
<form method=post>
    <input type=text name=key>
    <input type=text name=val>
    <input type=submit>
<h1>#{show sess}
|]

postRoot :: Handler ()
postRoot = 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)
    redirect Root

instance Yesod SessionExample where -- Set session timeout to 1 minute
    clientSessionDuration _ = 1

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

Messages

Post/Redirect/Get-Problem: Z.B. nach erfolgreichem Ausüllen eines Formulars den Benutzer umleiten und gleichzeitig eine kurze, einmalige Status Botschaft übermitteln.

```
setMessage :: Html -> GHandler sub master ()
```
Setzt eine Message in der Sitzung.

```
getMessage :: GHandler sub master (Maybe Html)
```
Liest letzte Message aus und löscht diese, sofern vorhanden.

`defaultLayout` sollte daher generell `getMessage` aufrufen und ggf. Botschaften darstellen

```
mkYesod "Messages" [parseRoutes|
/ RootR GET
/set-message SetMessageR POST
|]

getRootR :: Handler RepHtml
getRootR = defaultLayout [whamlet|
<form method=post action=@{SetMessageR}>
    My message is: #
    <input type=text name=message>
    <input type=submit>
|]

postSetMessageR :: Handler ()
postSetMessageR = do
    msg <- runInputPost $ ireq textField "message"
    setMessage $ toHtml msg
    redirect RootR

instance Yesod Messages where
    defaultLayout widget = do -- Demo, nicht notwendig
        pc <- widgetToPageContent widget
        mmsg <- getMessage
        hamletToRepHtml [hamlet|
$doctype 5
<html>
    <head>
        <title>#{pageTitle pc}
        ^{pageHead pc}
    <body>
        $maybe msg <- mmsg
            <p>Your message was: #{msg}
        ^{pageBody pc}
|]
```

Ultimate Destination

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

```
setUltDest :: RedirectUrl master url =>
              url -> GHandler sub master ()
```
Setzt Ultimate Destination.

```
setUltDestCurrent :: GHandler sub master ()
```
Setzt Ultimate Destination auf aktuelle Seite, falls $\neq$ 404.

```
setUltDestReferer :: GHandler sub master ()
```
Setzt Ultimate Destination auf Referer, falls ungesetzt.

```
redirectUltDest :: RedirectUrl master url =>
                   url -> GHandler sub master a
```
Führt einen Redirect aus und löscht Ultimate Destination.

```
clearUltDest :: GHandler sub master ()
```
Löscht Ultimate Destination.

```
mkYesod "UltDest" [parseRoutes|
/ RootR GET
/setname SetNameR GET POST
/sayhello SayHelloR GET
|]

getSetNameR = defaultLayout [whamlet|
<form method=post>
    Mein Name lautet #
    <input type=text name=name>
    . #
    <input type=submit value="Namen abschicken">
|]

postSetNameR :: Handler ()
postSetNameR = do
    name <- runInputPost $ ireq textField "name" -- Get name and set in session
    setSession "name" name
    redirectUltDest RootR   -- After setting name, redirect to the ultimate destination
                            -- If no destination is set, default to the homepage
getSayHelloR = do
    mname <- lookupSession "name"        -- Lookup name value set in session
    case mname of
        Nothing -> do                    -- No name in the session
            setUltDestCurrent            -- set current page as ultimate destination
            setMessage "Wie heisst Du?"
            redirect SetNameR            -- redirect to SetName page
        Just name -> defaultLayout [whamlet|
<p>Hallo #{name}
|]

getRootR = defaultLayout [whamlet|
<p>
    <a href=@{SetNameR}>Namen angeben
<p>
```

Generalised Algebraic Datatypes (GADTs)

```
data Expr = I Int                  -- integer constants
          | B Bool                 -- boolean constants
          | Add Expr Expr          -- add two expressions
          | If Expr Expr Expr      -- conditional
```

Definiert die Konstruktoren:

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

Alle Konstruktoren eines herkömmlichen Datentyps mussen den selben Ergebnistyp haben.

GADTs heben diese Einschränkung auf!

Generalised Algebraic Datatypes (GADTs)

```
data Expr a where
  I   :: Int  -> Expr Int
  B   :: Bool -> Expr Bool
  Add :: Expr Int -> Expr Int -> Expr Int
  If  :: Expr Bool-> Expr Int -> Expr Int -> Expr Int
```

Pattern Matching as usual:

```
eval :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (If eb e1 e2)
  | (eval eb) == True  = eval e1
  | (eval eb) == False = eval e2
```

Anwendungsbeispiel: Typ-sichere DSLs

Persistenz

Manchmal sollen Daten länger als eine Sitzung halten.

`Database.Persist` und `Yesod.Persist` stellen dazu eine Datenbank-Schnittstelle bereit.

- `Persistent` unterstützt verschiedene Datenbanken: PostgreSQL, SQLite, MongoDB, CouchDB, MySQL
- `Persistent` ist non-relational
- `Persistent` führt viele SQL Migrationen automatisch aus

Im Gegensatz zu Session-Maps ist die Datenbank-Schnittstelle im Typ-sicher, auch wenn die Datenbank selbst ungetypt ist.

Kein Unit-Tests notwendig!

Typ-Sichere Persistenz

Datenbanken werden mithilfe von TemplateHaskell spezifiziert:

```
mkPersist sqlSettings [persist|
Person
    name String
    age Int
    deriving Show
BlogPost
    title String
    authorId PersonId
|]
```

Definiert Hilfsfunktionen und die Haskell-Datentypen:

```
data Person { name :: String, age :: Int }
  deriving (Show, Read, Eq)
type PersonId = Key SqlPersist Person

data BlogPost { title :: String, authorId :: PersonId }
  deriving (Read, Eq)
```

```
type PersonId = Key SqlPersist Person

instance PersistEntity Person where -- Generalized Algebraic Datatype (GADT)
    data EntityField Person typ where
        PersonId   :: EntityField Person PersonId
        PersonName :: EntityField Person String
        PersonAge  :: EntityField Person Int
    type PersistEntityBackend Person = SqlPersist

    toPersistFields (Person name age) = [SomePersistField name, SomePersistField age]

    fromPersistValues [nameValue, ageValue] = Person
        <$> fromPersistValue nameValue
        <*> fromPersistValue ageValue
    fromPersistValues _ = Left "Invalid fromPersistValues input"

    -- Information on each field, used internally to generate SQL statements
    persistFieldDef PersonId = FieldDef
        (HaskellName "Id")
        (DBName "id")
        (FTTypeCon Nothing "PersonId")
        []
    persistFieldDef PersonName = FieldDef
        (HaskellName "name")
        (DBName "name")
        (FTTypeCon Nothing "String")
        []
    persistFieldDef PersonAge = FieldDef
        (HaskellName "age")
        (DBName "age")
        (FTTypeCon Nothing "Int")
        []
    data Unique Person typ = IgnoreThis
    entityDef = undefined
    halfDefined = undefined
```

Benutzerspezifische Datenbankfelder

Feldtypen müssen Instanzen der Klasse `PersistField` sein.

Für Enumerations kann die Instanz-Deklaration abgeleitet werden:

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

mkPersist sqlSettings [persist|
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:

```
mkPersist sqlSettings [persist|
Person
    firstName String
    lastName String
    age Int
    UniquePerson firstName lastName
    deriving Show
|]
```

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

Nachschalgen mit `getBy`:

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

Attriute

```
mkPersist sqlSettings [persist|
Person
    firstName String
    lastName String
    age Int Maybe
    created UTCTime default=now()
    deriving Show
|]
```

- `Maybe` wird hinten angestellt und macht das Feld optional in der Datenbank nullable
  Achtung: Uniqueness funktioniert nur mit nicht-optionalen Feldern, da SQL nicht beantwortet ob `NULL==NULL` gilt
  Haskell: Ja, PostgreSQL: Nein
- Mit `default` können Standardwerte vorgegeben werden
  Achtung: Dies ist Datenbank spezifisch!

Datenbank Schnittstelle

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

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

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

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

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

```
main = withSqliteConn ":memory:" $ runSqlConn $ do
  runMigration $ migrate entityDefs (undefined :: Person)
  steffenId <- insert $ Person "Steffen" 37
  steffen <- get steffenId
  liftIO $ print steffen
```

Datenbank Schnittstelle

```
main = withSqliteConn ":memory:" $ runSqlConn $ do
  runMigration $ migrate entityDefs (undefined :: Person)
  steffenId <- insert $ Person "Steffen" 37
  steffen <- get steffenId
  liftIO $ print steffen
```

withSqliteConn
: Anbindung an Datenbank je nach String Argument

runSqlConn Datenbank Aktionen ausführen; genau eine Datenbank Transaktion pro Aufruf von `runSqlConn`
: Alle Aktionen werden zurückgenommen, wenn bei einer Aktion darin eine Ausnahme auftritt

runMigration
: Default-Migration; auch beim Erstellen der Datenbank

Datenbank Migration

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

Automatische Migration bei:

- Zusätzliche Datentypen hinzufügen
- Zusätzliche Felder mit Default hinzufügen
- Typwechsel bei Felder, sofern Konversion möglich

Manuelle Intervention notwendig bei: `runMigrationUnsafe`

- Felder löschen
- Umbenennung von Felder oder Datentypen

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

Datenbank Abfragen

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

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

liefert Ergebnis-List

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

liefert nur das erste Ergebnis

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

liefert nur die Schlüssel der Ergebnisse

Direkte, rohe, typ-unsichere SQL Operationen sind auch möglich.

Datenbank Manipulationen

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

Datenbankwert verändern

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

nur spezielle Werte verändern

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

nur spezielle Werte löschen

```
personId <- insert $ Person "Andreas" "Abel" 37
update personId [PersonAge =. 38]

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

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

Datenbanken Integration in Yesod

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

```
runDB :: YesodDB sub master a -> GHandler
```

Datenbank Zugriff in Handler Funktion

```
get404 :: <...> =>
          Key b val -> b 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 Build Tool kümmert sich bereits um alles Wesentliche!

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persist|
Person
    firstName String
    lastName String
    age Int Gt Desc
    deriving Show
|]

data PersistTest = PersistTest ConnectionPool
mkYesod "PersistTest" [parseRoutes|
/person/#PersonId PersonR GET
|]
instance Yesod PersistTest
instance YesodPersist PersistTest where
    type YesodPersistBackend PersistTest = SqlPersist

    runDB action = do
        PersistTest pool <- getYesod
        runSqlPool action pool

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

openConnectionCount :: Int
openConnectionCount = 10

main :: IO ()
main = withSqlitePool "test.db3" openConnectionCount $ \pool -> do
    runSqlPool (runMigration migrateAll) pool
    runSqlPool (insert $ Person "Steffen" "Jost" 37) pool
    runSqlPool (insert $ Person "Andreas" "Abel" 38) pool
    warpDebug 3000 $ PersistTest pool
```

Weitere Yesod Themen:

- Authentifizierung
- Wechsel von Entwicklung zum produktiven Webserver
- Internationalisierung
- Mastersites and Subsites

Literatur

- "Developing Web Applications with Yesod and Haskell"
  by Michael Snoyman (O'Reilly, 2012)
- www.yesodweb.com Yesod Website
- A Gentle Introduction to Haskell
  by Paul Hudak, John Peterson and Joseph Fasel
- The Haskell Wiki