LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

**Fortgeschrittene Funktionale Programmierung**
Sommersemester 2020

# Anwendungsprogrammierung in Haskell: Webprogrammierung mit Yesod und GUI-Programmierung mit Gtk2Hs und Threepenny-GUI


Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 22. Juli 2020 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei


## Einleitung

Ziel des Kapitels: Einblick geben in das Erstellen von Real-World-Programmen:

- Webprogrammierung mit Yesod
  - Datenbank-Anbindung mit Persistent
- GUI-Programmierung mit Gtk2Hs
- GUI-Programmierung mit Threepenny-GUI (nur ein Beispiel)

Es gibt viele weitere Ansätze und Frameworks für diese Anwendungen, diese können natürlich auch für das Abschlussprojekt verwendet werden.



# Webprogrammierung mit Yesod



## Webframeworks, Yesod

- Übersicht über Webframe-Frameworks für Haskell: `https://wiki.haskell.org/Web/Frameworks`.
- Yesod: Wird z.B. für Uni2work verwendet
- Wir geben nur eine Überblick, Details sind im Yesod-Buch oder auf den Yesod-Webseiten zu finden `http://yesodweb.com`

Wesentliche Eigenschaften von Yesod:

- Typ-sichere URLs
- Templating / DSLs, d.h. viele modulare Einzelteile,
- integrierte Datenbankanbindung über `persist` und `conduit`
- REST-Architektur = REpresentational State Transfer
  d.h. zustandslos: gleiche URL bezeichnet stets die gleiche Webseite

## Installation

Siehe auch: `https://www.yesodweb.com/page/quickstart`
Vorgefertigtes Projekt erzeugen (Scaffolding):

```
> stack new mein-projekt yesodweb/sqlite
```

(mit `stack templates` weitere Templates anschauen)

Danach

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

Nun läuft lokaler Development-Webserver: `http://localhost:3000/`



## Anmerkungen zum Scaffolding

- Die vorgefertigten Templates stellen Vieles vorab ein
- Auslagern verschiedener Dinge in separate Dateien
- Wir werden für die Beispiele fast immer selbsterstellte Singe-Quelltextdateien verwenden
- Übertragung auf das Template ist aber einfach möglich.

Tool `yesod` bietet weitere Untersützung z.B.

`yesod add-handler` bietet interaktive Unterstützung zum Anlegen eines neuen Handlers

`yesod devel`: Webserver wird bei Dateiänderungen autom. neu kompiliert



## Hello-Yesod-Programm

```
-- notwendige Spracherweiterungen:
{-# LANGUAGE OverloadedStrings, TypeFamilies, TemplateHaskell, QuasiQuotes #-}

module Main where
import Yesod

data MyApp = MkApp
instance Yesod MyApp                         -- App-Instanz

                                             -- Routen festlegen (QuasiQuoter!)
mkYesod "MyApp" [parseRoutes|
  / HomeR GET
  |]

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

main :: IO ()
main = warp 3000 MkApp                      -- Webserver am Port 3000 starten
```



## Shakespearean Templates

Verwenden TemplateHaskell / QuasiQuoter um HTML, CSS, JavaScript zu manipulieren:

| Sprache | Quasiquoter | Dateiendung |
|---|---|---|
| HTML | `hamlet` | .hamlet |
| CSS | `cassius` | .cassius |
| CSS | `lucius` | .lucius |
| JavaScript | `julius` | .julius |
| I18n interpolierter Text | `stext/ltext` | |

Beispiel:

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

erzeugt Haskell-interne Darstellung von HTML, aus einer vereinfachten HTML-Syntax.

Scaffolding: Templates über Meta-Funktion aus Datei laden: `$(widgetFile` *Filename*`)`

## Interpolation

Wie Splicen für Haskell-Code, kann man in den Templates Interpolieren:

- `#{ }` Variablen-Interpolation von Variablen im Scope
  (escaped, automatisches Aufrufen von `toHtml`)
  Erlaubt: Haskell-Funktionen anwenden, Strings, Zahlen, qualifizierte Modulnamen
- `@{ }` für die typsichere URL-Interpolation, z.B. `@{HomeR}`.
- `^{ }` für die Template-Einbettung, fügt ein Template gleichen Typs ein.
- `_{ }` für internationalisierten Text, fügt eine Übersetzung ein.

Dynamisches Erzeugen damit möglich:

```
[whamlet|
  Value of fib22 is #{show (fib 22)}
|]
```

Der Ergebnistyp einer Interpolation muss immer eine Instanz der Typklasse `ToHtml` sein.



## Interpolation (2)

Interpolation von URLs:

```
let foo = show (fib 22) in
      [whamlet|
        Value of foo is #{foo}
        Return to <a href=@{Home}>Homepage
        .
      |]
```

`Home` ist Konstruktor für das Routing der Webanwendung.



## Hamlet

Hamlet = HTML + Interpolation, aber:

Schließende HTML-Tags werden durch Einrücken ersetzt.

| HTML-Code | In Hamlet |
|---|---|
| `<p>Some paragraph.`<br>`  <ul><li>Item 1</li>`<br>`      <li>Item 2</li>`<br>`  </ul></p>`<br>`<p>Next paragraph.</p>` | `<p>Some paragraph.`<br>`  <ul>`<br>`    <li>Item 1`<br>`    <li>Item 2`<br>`<p>Some paragraph.` |

QuasiQuoter `[hamlet|...]` erzeugt das gezeigtt HTML vom Typ `Html`.

Es sind jedoch kurze geschlossene inline-Tags sind zulässig:

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



## Hamlet (2)

HTML-Attribute: Wie in HTML, d.h. Gleichheitszeichen, Wert und Anführungszeichen sind meist optional. Abkürzungen für IDs, Klassen und Konditionale sind 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: Konstrukte

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>#{show person}
```



## Hamlet: Konstrukte (2)

Maybe-Konstrukt:

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

Auch Pattern-Matching und unvollständige Fälle:

```
$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: Konstrukte (3)

`$with` ist ähnlich zum `let`

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

Es gibt weitere Abkürzungen, z.B. `$doctype 5` für `<!DOCTYPE html>`.



## Lucius

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

Beispiel:

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

Normales CSS

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

Geschachtelt und mit Variablen

## Cassius

- Alternative zu Lucius
- Whitespace-sensitiv, dafür ohne Klammern und Semikolon

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



## Julius

Julius akzeptiert gewöhnliches JavaScript, jedoch erweitert um

- `#{}` für die Variablen-Interpolation,
- `@{}` für die URL Interpolation und
- `^{}` für das Template Embedding von anderen JavaScript Templates

Sonst ändert sich nichts, auch nicht an Einrückungen.



## Widgets

Widgets fassen einzelne Templates der 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/jquery.min.js"
  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 zum Kombinieren dieser Bausteine
- Alles wird automatisch dahin sortiert, wo es hingehört



## Template Embedding

`whamlet` bzw. `.whamlet`-Dateien erlauben 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!
      |]
```

Namenskonflikte verhindern durch dynamische IDs:

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

## Foundation-Typ

- Einstellungen einer Yesod Applikation: Mit einem Foundation-Datentyp
- Wert muss nichts speichern (kann aber)
- Ausreichend für Standardeinstellungen:

```
data MyWebApp = MkWebApp       -- Foundation Type
instance Yesod MyWebApp        -- Defaults
```

- Klasse Yesod fasst viele Einstellungen zusammen, z.B. das Rendern/Parsen von URLs, Funktion defaultLayout für Layout Webseiten, Authentifizierung, Sitzungsdauer, Cookie-Handling, Fehlerbehandlung, Aussehen der Fehlerseiten, externe CSS, Skripte und statische Dateien, usw.
- Bei Bedarf überschreiben:

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



## Foundation-Typ (2)

- Foundation-Typ mit Parameter möglich
- Kann in Handler-Monade mit getYesod ausgelesen werden.

```
data MyWebApp = MkWebApp { intPar :: Int
                         , varPar :: TVar String }
instance Yesod MyWebApp

getHomeR :: Handler Html
getHomeR = defaultLayout $ do
    master <- getYesod                           -- master :: MyWebApp
    let myint = intPar  master                   -- myint  :: Int
    mystr <- liftIO $ readTVarIO $ varPar master -- mystr  :: String
    toWidget [whamlet|
              #{mystr}
             |]
    mkYesod "MyWebApp" [parseRoutes|
                        / HomeR GET
                       |]

main = do v <- newTVarIO "Hello"
          warp 3000 MkWebApp {intPar = 0, varPar = v}
```



## Routen und Handling

DSL zur Spezifikation, QuasiQuoter parseRoutes

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

Format besteht aus drei Teilen:

1. Pfad
   - Statische Pfade wie /blog/help
   - Dynamische Pfade:/#<Typ> Typ benötigt Instanzen für PathPiece,Eq,Read,Show
   - Dynamische Multipfade: /*<Typ> Typ benötigt Instanzen für PathMultiPiece,Eq,Read,Show
2. Konstruktor für die typsichere URL
3. Erlaubte Protokolle oder Foundation-Typ und Konverter für Subsite



## PathPiece / PathMultiPiece

Klassen PathPiece und PathMultiPiece aus dem Modul Yesod.Dispatch legen Parser fest:

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

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

Vordefiniert:

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

## Überlappende Routen

- Yesod kann nicht inferieren, dass `/blog/help` und `/blog/#Int` nicht überlappen
- Überlappende Routen wie etwa `/blog/#Int` und `/blog/#Text` erzeugen eine Fehlermeldung
- Kann mit `!` am Anfang verhindert werden, z.B. `!/blog/#Text`
- Die von oben-nach-unten zuerst geparste Route wird eingeschlagen



## Routen und Handling

DSL zur Spezifikation, QuasiQuoter `parseRoutes`

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

Format besteht aus drei Teilen.

1. Pfad
2. Konstruktor für die typsichere URL
   - Erzeugt Deklaration für den Datentyp `Route <FoundationType>`
   - kann in der URL-Interpolation `@{ }` verwendet werden, z.B. bei dynamischen Pfaden: `@{BlogPostR 7}`).
3. Erlaubte Protokolle oder Foundation-Typ und Konverter für Subsite



## Routen und Handling

DSL zur Spezifikation, QuasiQuoter `parseRoutes`

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

Format besteht aus drei Teilen.

1. Pfad
2. Konstruktor für die typsichere URL
3. Erlaubte Protokolle oder Foundation-Typ und Konverter für Subsite
   - Erlaubte HTTP-Anfragen: GET, POST, PUT, DELETE, ...
   - keine Einschränkung: alle Anfragen erlaubt
   - oder Foundation-Typ der Subsite und Funktion zur Umrechnung der Foundation-Typen. Static wie im Beispiel wird durch Scaffolding angelegt.



## Handler

```
[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.
Diese Funktion behandelt die Route. Der Name der Funktion muss `<AnfrageTyp> ++ <Resource>` sein, z.B.

```
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.

## Handling

Handler-Funktionen sind in der Handler-Monade, meist vom Typ `Handler Html`.

```
type Handler a = HandlerFor App (IO a)
data HandlerFor site a = ...
```

`Handler`= Datentyp mit Foundation-Typ (`site`) und Rückgabewert der Monade (`a`)

Statt `Html` auch CSS, JSON, usw. möglich, muss Instanz von `ToTypedContent` sein

Auch möglich: unterschiedliche Repräsentationen über eine URL abwickeln:

```
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 = "Horst" :: Text
  age  = 40 :: Int
```

Je nach Anfrage wird HTML ausgeliefert oder JSON.



## Wichtige Funktionen der Handler-Monade

- `getYesod`: Wert des Foundation-Typs auslesen
- `getUrlRender`: Renderer für Werte des `Route`-Typs erhalten
- `getRequest`: Anfrage im Roh-Format erhalten
- `liftIO`: Ausführen von IO-Aktionen.
- `sendFile`: eine Datei versenden.
- `addHeader`: Antwort-Header festlegen.
- `redirect`: Umleitung zu anderer Ressource
- `notFound`, `permissionDenied`: explizite Fehlermeldungen
- `setCookie`, `lookupCookie`: Cookies bearbeiten.

`Handler` ist Instanz von `MonadLogger`:

Erzeugen von Log-Messages innerhalb von Templates geht mit `$logError`, `$logWarn`, `$logInfo` und `$logDebug`



## Webformulare

Drei Varianten:

- Applikatives Interface: einfach zu programmieren, Kontrolle über Aussehen ist eingeschränkt
- Monadisches Interface: flexibel gestaltbare Formulare, Verwendung komplizierter
- Input-Interface: keine eigene HTML-Darstellung, Verwendung mit bestehenden Formularen

Konvention: Funktionen beginnen mit `a`, `m` oder `i`, je nach Formulartyp, z.B. `areq` und `mopt`



## Applikative Formularfelder

Beispiel:

```
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
```

- `areq`: erforderliche Felder
- `aopt`: optionale Felder (`Maybe`-Typ)
- 1. Argument: Typ des Eingabefelds, erklärt den Parser, viele vordefinierte Felder
- 2. Argument: Objekt vom Typ `FieldSettings` (Bezeichner, Tooltip, id- und name-Attribut), im Beispiel wird mit `OverloadedStrings` nur Bezeichner gesetzt
- 3. Argument: Default-Wert (optional, dahe Maybe)

## Formulare erstellen

`AForm` in `MForm` umwandeln mit `renderTable`, `renderDivs`, oder `renderBootstrap`

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

Anschließend kann es wie folgt verwendet werden:

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

Formular erzeugen mit `generateFormGet` oder `generateFormPost` je nach HTTP-Methode
`form`-Tag und Knopf zum Absenden einfügen



## Formulare auswerten

Auswerten mit `runFormGet` bzw. `runFormPost`

```
postCarR :: Handler Html
postCarR = do ((result,widget), enctype) <- runFormPost carForm
              case result of
                FormSuccess car -> defaultLayout [whamlet|
                    <h2>Car received:
                    <p>#{show car}  |]
                FormMissing -> addMessage "i" "No data" >> redirect CarR
                FormFailure msgs  -> defaultLayout [whamlet|
                    <h2>Fehler:
                    <ul>
                        $forall msg <- msgs
                            <li>#{msg}
                    <form method=post action=@{CarR} enctype=#{enctype}>
                        ^{widget}
                        <button>Submit |]
```

Mögliche Ergebnisse für `result`:

- `FormSuccess a` bedeutet Erfolg
- `FormFailure [Text]`: Parsen fehlgeschlagen
- `FormMissing`: Keine Daten vorhanden



## Auswahllisten

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

```
data Car   = Car {carModel::Text, carYear::Int, carColor::Maybe Color}
        deriving Show
data Color = Red | Blue | Gray | Black deriving (Show, Eq, Bounded, Enum)

carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt (selectFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue), ("Grau", Gray), ("Schwarz", Black)]
```



## Auswahlknöpfe

`radioFieldList` nimmt eine Liste von (`Text`,Wert)-Paaren und erzeugt Auswahlknöpfe. Z.B.

```
carAForm :: AForm Handler Car
carAForm = Car
    <$> areq textField "Model" Nothing
    <*> areq intField  "Year"  (Just 1994)
    <*> aopt (radioFieldList colors) "Color" Nothing
  where
    colors :: [(Text, Color)]
    colors = [("Rot", Red), ("Blau", Blue), ("Grau", Gray), ("Schwarz", Black)]
```

## Input-Formulare: Direkte Verwendung von HTML-Formularen

```
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}|]
```

Nachteil: Übereinstimmung der Name-Tags wird nicht geprüft.

`ireq` und `iopt` haben nur noch zwei Argumente, den Feld-Typ und den Feld-Namen.

Wenn Daten nicht passen, erfolgt eine Umleitung auf eine „Invalid Arguments"-Fehlerseite.



## Monadische Formulare

Monadische Formulare: eigenes Layout, kümmern sich um einzigartige Name-Tags, usw.

```
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 (2)

- `extra` muss irgendwo ins Formular eingebaut werden.
  Bei GET-Formulare signalisiert es Absenden des Formulars
  Bei POST-Formularen Verhinderung von Cross-Site-Request-Forgery Angriffen.
- Felder mit `mreq` und `mopt` erzeugen:

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

- `mreq` und `mopt` funktionieren analog zu `areq` und `aopt`, aber Namen der Eingabefelder werden ignoriert, denn das Layout wird explizit angegeben.
- Felder werden durch `FormResult` und `FormView` beschrieben.
- `FormResult` zum Parsen
- `FieldView`: Record mit `fvLabel`, `fvTooltip`, `fvId`, `fvInput`, `fvErrors` und `fvRequired`.
- Z.B. `fvInput` erzeugt Input-Feld aus FieldView.
- `fvId` Zugriff auf Id.



## Sessions

- HTTP kennt keinen Zustand, Abhilfe: Sessions
- Kleine Menge an Daten (z.B. eine Sitzung-ID) werden mit jeder Anfrage übermittelt
- skaliert gut mit mehreren Servern, da jeder Request in sich abgeschlossen ist
- keine zentrale Koordination/Datenbank notwendig
- Yesod: Verschlüsselung und Signatur der Sitzungsdaten
- Sitzung verfallen automatisch nach 2 Stunden.
- Einstellbar über Yesod-Instanz des Foundation-Typs:

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

## Sessions (2)

Eine Sitzung ist eine ungetypte Map:

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

Wesentliche Operationen:

- `getSession::MonadHandler m => m SessionMap` liefert die gesamte Sitzungs-Map
- `lookupSessionBS::MonadHandler m=>Text->m (Maybe ByteString)` Schlüssel nachschlagen
- `lookupSession::MonadHandler m => Text->m (Maybe Text)` Schlüssel nachschlagen
- `setSession::MonadHandler m => Text -> Text -> m ()` Schlüssel-Wert-Paar setzen
- `deleteSession::MonadHandler m => Text -> m ()` Schlüssel löschen
- `clearSession::MonadHandler m => m ()` Gesamte Sitzungs-Map löschen



## Sessions-Beispiel

Hinzufügen und Löschen von einzelnen Schlüssel/Wert-Paaren

```
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

- Ausfüllen von Formularen: Problem: Feedback des Erfolgs an Nutzer
- Lösung: Jede Seite prüft, ob spezielles Sitzungs-Feld für Nachrichten existiert
- Yesod bietet eigene Schnittstelle dafür:
  - `addMessage :: MonadHandler m => Text -> Html -> m ()` setzt Message
  - `getMessages :: MonadHandler m => m [(Text, Html)]` liest Messages aus und löscht
- `defaultLayout` nutzt `getMessages`, um ggf. Botschaften anzuzeigen.



## Messages: Beispiel

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

getB = do msgs <- getMessages
          page <- defaultLayout $ do
                    [whamlet|
                       <p>You are at B
                       $forall (cls,msg) <- msgs
                          <p class="#{cls}">Message: #{msg}
                    |]
          addMessage "warning" "Previous: B"
          return page
```

## Persistenz – Anbindung an Datenbanken

- Dauerhaftes Speichern von Daten
- Bibliotheken `Database.Persist` und `Yesod.Persistent` stellen *typsichere* Schnittstelle für Standard-Datenbanken bereit
- `Persistent` unterstützt verschiedene Datenbanken, u.a. SQLite, PostgreSQL, MySQL, MongoDB
- `Persistent` führt viele SQL Migrationen automatisch aus
- `Database.Persist` unabhängig von Yesod



## Beispiel

```
{-# LANGUAGE EmptyDataDecls, FlexibleContexts, GADTs, GeneralizedNewtypeDeriving       #-}
{-# LANGUAGE MultiParamTypeClasses, OverloadedStrings, QuasiQuotes, TemplateHaskell     #-}
{-# LANGUAGE TypeFamilies, DerivingStrategies, StandaloneDeriving. UndecidableInstances #-}
import Control.Monad.IO.Class (liftIO)

import Database.Persist
import Database.Persist.Sqlite
import Database.Persist.TH
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 fIrst 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]
```



## Spezifikation der Datenbanken

Mit QuasiQuoter und TemplateHaskell:

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

Definiert Hilfsfunktionen und die 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)
```



## Spezifikation der Datenbanken (2)

- Feldtypen müssen Instanzen der Klasse `PersistField` sein.
- Für Enumerations automatisierbar mit TemplateHaskell:

```
data Employment = Employed | Unemployed | Retired
    deriving (Show, Read, Eq)
derivePersistField "Employment"
-- Andere Datei:
share [mkPersist sqlSettings, mkMigrate "migrateAll"]
[persistLowerCase|
Person
    name String
    employment Employment
|]
```

## Spezifikation der Datenbanken (3)

- Konstruktoren werden in der Datenbank als String gespeichert
- Erlaubt nachträgliche Erweiterung der Konstruktoren.
- Zeilen, die mit Großbuchstaben beginnen: Einzigartigkeit von Datenbankeinträgen, z.B.

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

- Erlaubt Nachschlagen Funktion `getBy`: `getBy $ UniquePerson "Horst"`
- Der eigentliche Schlüssel bleibt unverändert.
- Mit `Primary firstName` echter Datenbankschlüssel



## Spezifikation der Datenbanken (4)

Optionale Felder erhalten zusätzlich ein `Maybe`

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



## Migration

- `Persist` kann sich um nachträgliche Änderungen an der Datenbank kümmern
- Z.B. Zeitstempel hinzufügen

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

- Automatische Migration `runMigration`: Datentypen hinzufügen, zusätzliche Felder mit Default hinzufügen, Typwechsel mit möglicher Konversion
- Felder löschen oder Felder umbenennen erfordert manuelle Intervention



## Datenbank-Schnittstelle

- Für SQLite:`runSqlite`
- Erstes Argument: Dateinamen oder `":memory:"`,
- Pro Aufruf von `runSqlite` eine Datenbanktransaktion

```
main = runSqlite ":memory:" $  do
  runMigration $ migrateAll -- ggf. Migration durchführen
  dieId <- insert $ Person "Horst" 37   -- Einfügen
  horst   <- get horstId                -- Abfragen
  liftIO $ print horst
```

- führt eine Default-Migration durch, sollte man mindestens einmal beim Erstellen einer neuen Datenbank durchführen

## Operationen (1)

Klasse `PersistStore b m` definiert u.a.

- `insert :: ... => val -> m (Key val)`: Wert in die Datenbank einfügen
- `get :: ... => Key b val -> m (Maybe val)`: Wert nachschlagen
- `getBy :: ... => Unique val -> m (Maybe (Entity val))`: `Unique` nachschlagen, Ergebnis vom Typ `Entity valId val`.
- `delete  :: ... => Key val -> m ()`: Löschen
- `repsert :: ... => Key val -> val -> m ()`: Einfügen oder ersetzen.



## Operationen (2)

Echte Datenbank-Abfragen:

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

- erhält Liste von Filtern und eine Liste von Auswahl-Optionen.
- Beispiel: Alle Personen im Alter zwischen 26 und 30:
  `people25bis30 <- selectList [PersonAge >. 25, PersonAge <=. 30] []`
- Filter sind UND-verknüpft.
- Übliche Operatoren aber mit Punkt am Ende, `!=.` anstelle von `/=.` verwendet.
- Die Operatoren `<-.` und `/<-.` stehen für „ist Element von“ und „ist kein Element von“.



## Operationen (3)

- Oder-Verknüpfungen müssen `||.` erzeugt werden.
- Z.B. alle Personen 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])
  )
  []
```



## Operationen (4)

Auswahl-Optionen

- `Asc` *`Feld`*: aufsteigend sortierte Ergebnisse,
- `Desc` *`Feld`*: absteigend sortierte Ergebnisse,
- `LimitTo n`: Ergebnis-Anzahl begrenzen
- `OffSetBy n`: die ersten $n$-Ergebnisse überspringen

Z.B.

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

## Operationen (5)

Abfragemöglichkeiten

- `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::...=>[Filter val]->Source(ResourceT (b m))(Key val)` (liefert nur die Schlüssel der Ergebnisse).



## Operationen (6)

Datenbank-Manipulationen:

- `update :: PersistEntity val => Key val -> [Update val] -> m ()` (Datenbankwert verändern),
- `updateWhere :: PersistEntity val => [Filter val] -> [Update val] -> m ()` (nur spezielle Werte verändern) und
- `deleteWhere :: PersistEntity val => [Filter val] -> m ()` (nur spezielle Werte löschen).
- mögliche Operatoren sind `=.`, `+=.`, `-=.`, `*=.` und `/=.`.

Z.B.

```
personId <- insert $ Person "Horst" "Hans" 39
update personId [PersonAge =. 40]
updateWhere [PersonFirstName ==. "Horst"] [PersonAge +=. 1]
```



## DB-Integration in Yesod

Datenbank/Yesod-Schnittstelle ist in `Yesod.Persist`

`runDB :: YesodDB site a -> HandlerFor site a`

Erlaubt Datenbankzugriff in der Handler-Monade.

Alternative zu `get`:

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

Bei Fehlschlagen: direkt eine 404-Fehlerseite

YesodPersist-Instanz des Foundation-Typs legt fest, welche DB verwendet wird



## Ein minimales Yesod-Beispiel

```
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
```

## Widgdets und Datenbankanfragen

Innerhalb von Widgets sind keine Datenbankanfragen erlaubt

Der folgende Code compiliert nicht:

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

Grund: Wir sind nicht mehr in der `Handler`-Monade

Abhilfe:

```
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}
  |]
```



## Mehrere Datenbankzugriffe

Mehrere Datenbankzugriffe möglichst zusammenfassen D.h. besser

```
getHomeR :: Handler Html
getHomeR do
   (p1Id,p2Id,list) <- runDB $ do
     pers1Id <- insert ...
     pers2Id <- insert ...
     list    <- selectList ...
     return (pers1Id, pers2Id, list)
```

anstelle von

```
getHomeR do
   pers1Id <- runDB $ insert ...
   pers2Id <- runDB $ insert ...
   list    <- runDB $ selectList ...
```

verwenden.



# GUI-Programmierung mit Gtk2Hs



## GUI-Programmierung mit Gtk2Hs

GUI-Programmierung in Haskell

- Viele Ansätze für die Programmierung grafischer Benutzeroberflächen
- Übersicht:. `https://wiki.haskell.org/Applications_and_libraries/GUI_libraries`
- Allgemeines Problem: Wartung der Softwarebibliotheken und Anpassung an neue Versionen

Gtk

- C-Framework Gtk (`https://www.gtk.org/`) (früher Gtk+),
- GIMP ToolKit, für Linux, Mac OS X, Windows, usw. verfügbar
- Verschiedene Bibliotheken:
  - GLib: Kernbibliothek, Kompatibilitätsschicht,Thread-Handling
  - Cairo: Bibliothek für 2D Vektor-Grafik
  - GDK: Rendern, Bitmaps
  - Pango: Textdarstellung, Internationalisierung
  - ATK: Zugänglichkeit, z.B. Vergrößern, Vorlesen
  - Glade: Grafisches GUI Design Tool

## GUI-Programmierung mit Gtk2Hs (2)

Gtk2Hs

- Anbindung von Haskell an Gtk
- Doku: `https://wiki.haskell.org/Gtk2Hs` (meistens Gtk+ 2.x!)
- Paket `gtk` ist Anbindung an *Gtk+ 2.x*
- Paket `gtk3` ist Anbindung an *Gtk 3.x*
- Paket `gi-gtk` andere Anbindung an *Gtk 3.x* (automatisch erstellt)

GUI-Designer Glade:

- GUI-Beschreibungen mit Glade 3.8 für Gtk+ 2.x
- Höhere Glade-Versionen: GUI-Beschreibungen für Gtk 3

Wir verwenden: Gtk2Hs via `gtk3`

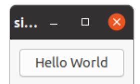


## Grundgerüst eines Gtk2Hs-Programms

- sehr imperativ, in der IO-Monade, ereignisgesteuert
- Gtk-Schleife fängt Ereignisse und ruft Callback-Funktionen auf
- Programmieren: Ansicht und Interaktion (durch Callback-Funktionen)

```
module Main where
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI                                              -- Initialisierung von Gtk
  window <- windowNew                                  -- Fenster erzeugen
  button <- buttonNew                                  -- Button erzeugen
  set window [ containerBorderWidth := 10,             -- Attribute setzen
               containerChild       := button ]
  set button [ buttonLabel := "Hello World"   ]
  on button buttonActivated callback)                  -- Ereignis an Callback-Fkt. binden
  on window objectDestroy mainQuit                     -- Ereignis an Callback-Fkt. binden
  widgetShowAll window                                 -- Widgets anzeigen
  mainGUI                                              -- Kontrolle an Gtk-Schleife

callback1 :: IO ()
callback1 = putStrLn "Hello World"
```

Erzeugt:

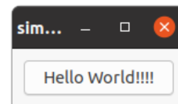





## Beispiel mit Auslesen der Attributwerte

```
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [ containerBorderWidth := 10,
               containerChild       := button ]
  set button [ buttonLabel := "Hello World"   ]
  on button buttonActivated $ callback2 button
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI

callback2 :: ButtonClass o => o -> IO ()
callback2 b = do
  l <- get b buttonLabel                -- GUI Call
  set b [ buttonLabel := (l ++ "!") ] -- GUI Call
```

nach 4x Button drücken






## Widgets

- Widget = Elemente einer GUI
- haben Attribute
- Modul `System.Glib.Attributes` definiert:

  `data ReadWriteAttr o a b`

  Attribute eines Objekts `o` mit Lesetyp `a` und Schreibtyp `b`
- Spezialfälle:

```
-- Gewöhnliches Attribut mit identischem Typ für beide Zugriffe
type Attr o a = ReadWriteAttr o a a

-- Attribut kann nur gelesen werden
type ReadAttr  o a = ReadWriteAttr o a ()}

-- Attribut kann nur geschrieben werden
type WriteAttr o b = ReadWriteAttr o () b}
```

## Lesen und Setzen von Attributen

```
get :: o -> ReadWriteAttr o a b -> IO a
set :: o -> [AttrOp o] -> IO ()
```

`AttrOp` verwendet existentielle Quantifizierung:

```
data AttrOp o  =
   forall a b.(:=)  (ReadWriteAttr o a b) b              -- Zuweisung eines Werts
 | forall a b.(:~)  (ReadWriteAttr o a b) (a -> b)       -- Anwendung einer Fkt. auf akt. Wert
 | forall a b.(:=>) (ReadWriteAttr o a b) (IO b)         -- Zuweisung des Werts einer IO-Aktion
 | forall a b.(:~>) (ReadWriteAttr o a b) (a -> IO b)    -- Anwendung einer IO-Fkt. auf akt. Wert
 | forall a b.(::=) (ReadWriteAttr o a b) (o -> b)       -- Anw. e. Fkt., die Objekt als Arg. bekommt
 | forall a b.(::~) (ReadWriteAttr o a b) (o -> a -> b) -- Anw. e. Fkt., die Objekt und Wert
                                                         -- als Arg. bekommt
```

Daher: Liste `[AttrOp o]` kann Elemente mit verschiedenen Typen für `a` und `b` haben:

```
set button [ buttonLabel        := "OK",
             buttonFocusOnClick := False ]
```



## Widgets (2)

`Graphics.UI.Gtk.Abstract.Widget` definiert

- `widgetShowAll :: WidgetClass self => self -> IO ()` zum Anzeigen eines vorbereiteten Widgets auf dem Bildschirm und
- `widgetDestroy :: WidgetClass self => self -> IO ()` zum dauerhaften Entfernen eines Widgets.

`Graphics.UI.Gtk.Abstract.Object` definiert das Signal

- `objectDestroy :: WidgetClass self => Signal self (IO ())`, welches beim Entfernen des Widgets ausgelöst wird.

Binden mit `on widget objectDestroy callback`



## Buttons

`Graphics.UI.Gtk.Buttons.Button` definiert u.a.

- `buttonNew :: IO Button`: Erzeugen eines Buttons
- `buttonNewWithLabel :: GlibString string => string -> IO Button`: Erzeugen mit gesetzter Beschriftung
- Attribute z.B. `buttonLabel` für die Beschriftung
- `buttonActivated :: ButtonClass self => Signal self (IO ())` Signal: Knopf gedrückt und wieder losgelassen

Binden mit
`on :: object -> Signal object callback -> callback -> IO (ConnectId object)`

Z.B.

`on button buttonActivated callback`



## Entries

`Graphics.UI.Gtk.Entry.Entry` definiert einzeilige Textfelder und

- `entryNew :: IO Entry`: neues Textfeld erzeugen
- `entrySetText :: ... => self -> string -> IO ()`: Text setzen
- `entryGetText :: ... => self -> IO string`: Text lesen
- Signale, z.B. `entryBackspace`: Drücken von Backspace im Eingabefeld
- Attribute, z.B. `entryEditable`

## Container-Widgets

Container-Widgets = Widgets, die Kinder-Widgets enhalten

`Graphics.UI.Gtk.Abstract.Container` definiert `Container` Typklasse `ContainerClass` mit:

```
containerAdd::(ContainerClass self, WidgetClass widget) => self->widget->IO ()
containerRemove::(ContainerClass self, WidgetClass widget) => self->widget->IO ()
containerGetChildren::ContainerClass self => self -> IO [Widget]
containerForeach::ContainerClass self => self -> ContainerForeachCB -> IO ()
type ContainerForeachCB = Widget -> IO ()
```

Beispiele:

- `Bin` = Container mit genau einem Kind
- `Window` = Fenster, Unterklasse von `Bin`



## Window

`Graphics.UI.Gtk.Windows.Window` definiert

- `windowNew :: IO Window`
- Attribute, z.B. `windowDefaultWidth` (Breite bei Initialisierung) und `windowTitle` (Fenstertitel).
- Funktion `windowPresent` schiebt ein Fenster nach vorne in der Ansicht.



## HBox

Container-Widget `HBox` dient zur horizontalen Ausrichtung (analog `VBox`)

`Graphics.UI.Gtk.Layout.HBox` definiert

- `hBoxNew :: Bool -> Int -> IO HBox`
- 1.Argument: `True`, wenn Kinder mit gleichmäßigem Platz angeordnet werden sollen
- 2.Argument: Anzahl an freien Pixeln zwischen Kindern

Anpassung der Kinder bei Größenänderungen der `HBox` über `data Packing` aus `Graphics.UI.Gtk.Abstract.Box`

- `PackGrow`: Kinder wachsen mit
- `PackNatural`: Kinder bestimmen eigene Größe selbst
- `PackRepel` Rand um die Kinder wächst

Einfügen der Kinder:

```
boxPackStart::(BoxClass self, WidgetClass child) => self->child->Packing->Int->IO ()
```

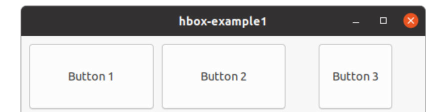
`Int`-Wert legt zusätzlichen Abstand fest.



## HBox: Beispiel

```
main = do
  initGUI
  window <- windowNew
  hbox     <- hBoxNew True 10
  button1 <- buttonNewWithLabel "Button 1"
  button2 <- buttonNewWithLabel "Button 2"
  button3 <- buttonNewWithLabel "Button 3"
  set window [ containerBorderWidth := 10,
               windowDefaultWidth    :=  500,
               windowDefaultHeight   :=  100,
               containerChild       := hbox ]
  boxPackStart hbox button1 PackGrow 0
  boxPackStart hbox button2 PackGrow 0
  boxPackStart hbox button3 PackRepel 0
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI
```

ergibt:



## Alignment

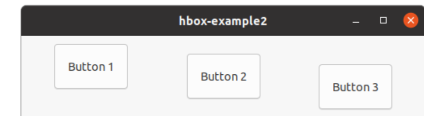
- Ausrichtung von Widgets nicht mit dem `Box`-Container möglich.
- Ausrichtung mit `Alignment`-Widgets aus `Graphics.UI.Gtk.Layout.Alignment`
- `alignmentNew :: Float -> Float -> Float -> Float -> IO Alignment`
- In `alignmentNew xalign yalign xscale yscale`
  `xalign` und `yalign` legen relative Ausrichtung fest
  (0=Links/Oben bis 1=Rechts/Unten; 0.5=Mitte)
  `xscale`, `yscale` legen relative Skalierung des Kindes fest
  (0=wächst nicht; 1=wächst voll).



## Alignment: Beispiel

```
main = do
  initGUI
  window <- windowNew
  hbox   <- hBoxNew True 10
  align1 <- alignmentNew 1 0 0.5 0.5
  align2 <- alignmentNew 0.5 0.5 0.5 0.5
  align3 <- alignmentNew 0 1 0.5 0.5
  button1 <- buttonNewWithLabel "Button 1"
  button2 <- buttonNewWithLabel "Button 2"
  button3 <- buttonNewWithLabel "Button 3"
  set window [ containerBorderWidth := 10,
               windowDefaultWidth    :=  500,
               windowDefaultHeight   :=  100,
               containerChild        := hbox ]
  containerAdd align1 button1
  containerAdd align2 button2
  containerAdd align3 button3
  boxPackStart hbox align1 PackNatural 0
  boxPackStart hbox align2 PackNatural 0
  boxPackStart hbox align3 PackNatural 0
  on window objectDestroy mainQuit
  widgetShowAll window
```

Von der Ausführung erzeugtes Fenster (unter Linux):






## Tabellen: Table

Table sind `Container`-Widgets, in `Graphics.UI.Gtk.Layout.Table`:

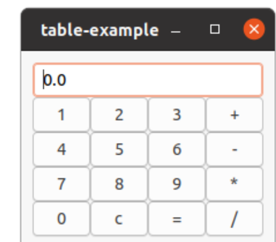
```
tableNew :: Int       -- Anzahl Zeilen
         -> Int       -- Abzahl Spalten
         -> Bool      -- Alle Felder gleich groß
         -> IO Table

tableAttachDefaults ::(TableClass self,WidgetClass widget)
  => self -> widget  -- Tabelle, einzufügendes Kind
  -> Int  -> Int     -- Linke und Rechte Spalte
  -> Int  -> Int     -- Obere und Untere Zeile
  -> IO ()
```



## Table: Beispiel

```
main = do
 initGUI
 window  <- windowNew
 buttons <- replicateM 16 buttonNew
 entry   <- entryNew
 table   <- tableNew 5 5 False
 set window [ containerBorderWidth := 10,
              containerChild       := table]
 entrySetText entry "0.0"
 set entry [entryEditable := False]
 sequence_ [do tableAttachDefaults table b i (i+1) j (j+1)
               set b [buttonLabel := lab]
           | (b,(i,j),lab) <- zip3 buttons
                                   [(j,i) | i <- [1..4], j <- [1..4]]
                                   ["1","2","3","+"
                                   ,"4","5","6","-"
                                   ,"7","8","9","*"
                                   ,"0","c","=","/"]]
 tableAttachDefaults table entry 0 5 0 1
 on window objectDestroy mainQuit
 widgetShowAll window
 mainGUI
```

erzeugt:

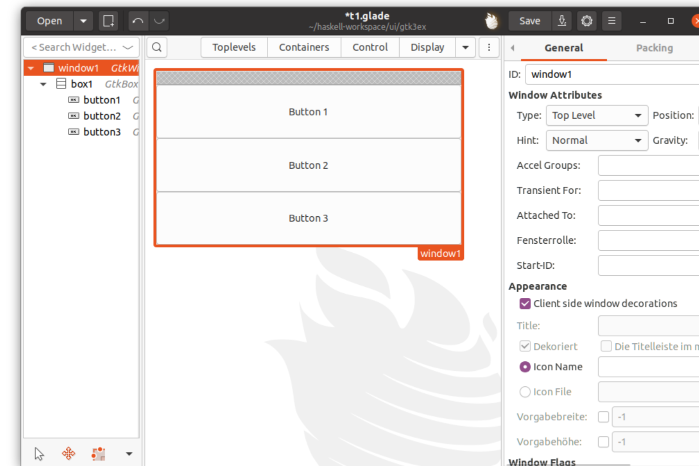


## Glade

- Glade ist ein Gui-Designer
- Die Optik kann damit direkt erzeugt werden
- Speichern als GUIBuilder-Format erzeugt XML-Beschreibung
- In Gtk2Hs: Laden der XML-Beschreibung oder Einbinden in den Quellcode



## Glade-Ansicht






## Glade: Verwendung

Modul `Graphics.UI.Gtk.Builder`:

- `builderNew :: IO Builder` Erzeugen eines leeren Builder-Objekts.
- `builderAddFromFile :: GlibFilePath fp => Builder -> fp -> IO ()`,
  Definition aus Datei zum Builder hinzufügen.
- `builderAddFromString :: Builder -> string -> IO ()`,
  Definition als String zm Builder hinzufügen.
- `builderGetObject:: ... => Builder -> (GObject -> cls) -> string -> IO cls`,
  um die Objekte wieder aus dem Builder zu extrahieren. Das zweite Argument ist ein Type-cast, z.B. `castToWindow`, `castToButton`, ... und das dritte Argument das Namen-Attribut des gesuchten Objektes in XML.

Einlesen des XML kann zu Laufzeitfehler führen!



## Glade: Beispiel

```
module Main where

import Graphics.UI.Gtkf
import Graphics.UI.Gtk.Builder

main :: IO ()
main = do
   initGUI
   builder <- builderNew
   builderAddFromFile builder "t1.glade"
   window <- builderGetObject builder castToWindow "window1"
   box1   <- builderGetObject builder castToBox "box1"
   button1 <- builderGetObject builder castToButton "button1"
   button2 <- builderGetObject builder castToButton "button2"
   button3 <- builderGetObject builder castToButton "button3"
   widgetShowAll window
   on button1 buttonActivated $ callback button1
   on button2 buttonActivated $ callback button2
   on button3 buttonActivated $ callback button3
   on window objectDestroy mainQuit
   mainGUI
```



```
  l <- get b buttonLabel              -- GUI Call
  set b [ buttonLabel := (l ++ "!") ] -- GUI Call
```

## Glade: Beispiel besser

Besser: Einen Datentyp mit allen Elementen am Anfang initialisieren

```
data MainGUI = MainGUI {  mainWindow :: Window
                       ,  mainBox    ::Box
                       ,  button1    :: Button
                       ,  button2    :: Button
                       ,  button3    :: Button      }
main = do initGUI
          gui <- loadGUI   -- selbstdefinierte Funktion
          ...

loadGUI = do -- all XML loading errors must occur here
  builder <- builderNew
  builderAddFromFile builder "t1.glade"
  mainWindow  <- builderGetObject builder castToWindow "window1"
  mainBox     <- builderGetObject builder castToBox    "box1"
  button1     <- builderGetObject builder castToButton "button1"
  button2     <- builderGetObject builder castToButton "button2"
  button3     <- builderGetObject builder castToButton "button3"
  return $ MainGUI mainWindow mainBox button1 button2 button3
```



## Eingefrorene GUI

```
module Main where
import Graphics.UI.Gtk
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [ containerChild := button ]
  set button [ buttonLabel := "Fib 1"    ]
  on button buttonActivated $ callback3 button
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI

callback3 :: ButtonClass o => o -> IO ()
callback3 b = do
  l1 <- get b buttonLabel
  let n   = read $ (words l1) !! 1
  let fin = show $ fib n
  let l2  = "Fib " ++ (show $ n+1) ++ " = " ++ fin
  set b [ buttonLabel := l2 ]
```

Problem:
GUI friert ein bei größeren Berechnungen in Callback-Funktion

Ansatz:
Callback-Fkt. in nebenl. Thread laufen lassen
reicht nicht, da Gtk nicht Thread-safe!

Verwende in der Callback-Fkt. zusätzlich

```
callback3 :: ButtonClass o => o -> IO ()
  l1 <- get b buttonLabel
  let n   = read $ (words l1) !! 1
  let fin = show $ fib n
  let l2  = "Fib " ++ (show $ n+1) ++ " = " ++ fin
  forkIO (seq fibn $
               (set b [ buttonLabel := l2 ]))
  return ()
```

führt Aktion im Hauptthread aus (blockiert aktuellen Thread)

- `postGUIAsync :: IO () -> IO ()` führt Aktion im Hauptthread aus (nicht blockierend)

```
callback3 :: ButtonClass o => o -> IO ()
```



## Menüs und Werkzeugleisten

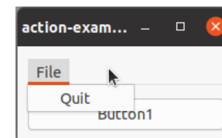
Menüs, Werkzeugleisten und Tastenkombinationen starten Aktionen vom Typ `Action`:

```
actionNew :: GlibString string =>
      string         --  name    : unique name for the action
  ->  string         --  label   : displayed in items & btns
  ->  Maybe string   --  tooltip
  ->  Maybe StockId  --  stockId: icon to be displayed
  ->  IO Action

actionActivated :: ActionClass self => Signal self (IO ())
```



## Menüs und Werkzeugleisten: Beispiel

```
module Main where
import Graphics.UI.Gtk
main :: IO ()
main = do
  initGUI
  window <- windowNew
  vbox    <- vBoxNew True 10
  createMenu vbox
  button <- buttonNewWithLabel "Button1"
  boxPackStart vbox button PackNatural 0
  set window [ containerBorderWidth := 10,
               containerChild    := vbox ]
  on window objectDestroy mainQuit
  widgetShowAll window
  mainGUI
```




```
-- XML-Menübeschreibung:
uiDecl = "\
\<ui>\
\  <menubar>\
\    <menu action=\"FILE_MENU\">\
\      <menuitem action=\"QUIT\"/>\
\    </menu>\
\  </menubar>\
\</ui>"

createMenu :: VBox -> IO ()
createMenu box = do
 actFileM <- actionNew "FILE_MENU" "File" Nothing Nothing
 actQuit <- actionNew "QUIT" "Quit" (Just "Exit") Nothing
 on actQuit actionActivated mainQuit
 actGroup    <- actionGroupNew "ACTION_GROUP"
 mapM (actionGroupAddAction actGroup) [actFileM,actQuit]
 ui <- uiManagerNew
 uiManagerAddUiFromString ui uiDecl
 uiManagerInsertActionGroup ui actGroup 0
 Just menubar <- uiManagerGetWidget ui "/ui/menubar"
 boxPackStart box menubar PackNatural 0
```

## Beispiel: Taschenrechner

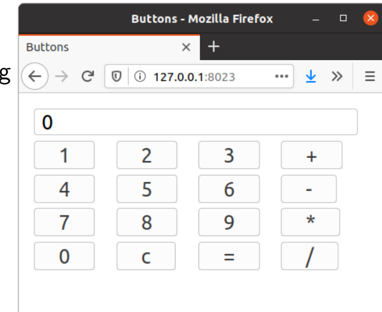
```
module Main where
import Graphics.UI.Gtk
import Control.Monad
import Data.Char
import Data.IORef
main = do
 initGUI
 window  <- windowNew
 buttons <- replicateM 16 buttonNew
 entry   <- entryNew
 let start  = (id, 0.0)
 cSt <- newIORef start
 let
  callback b = do l:_ <- get b buttonLabel
                  calcStep l
                  x <- readIORef cSt
                  entrySetText entry (show (snd x))
  calcStep x | isDigit x
    = digit (fromIntegral $ digitToInt x)
  calcStep o | o `elem` "+-*/" = oper o
  calcStep '=' = total
  calcStep 'c' = clear
  calcStep  _  = return ()
  oper op = modifyIORef cSt (\(f,n)->(op (f n),0))
  total = modifyIORef cSt (\(f,n)->(id,f n))
  clear = modifyIORef cSt (\(f,n)->if n == 0.0
                         then start else (f,0.0))
  digit i = modifyIORef cSt (\(f,n)->(f,n*10+(fromIntegral i)))
  ...
```

```
...
table   <- tableNew 5 5 False
set window [containerBorderWidth := 10
           ,containerChild := table]
entrySetText entry "0.0"
set entry [entryEditable := False]
sequence_ [do tableAttachDefaults table b i (i+1) j (j+1)
              set b [buttonLabel := lab]
         | (b,(i,j),lab) <-
             zip3 buttons
                  [(j,i) | i <- [1..4], j <- [1..4]]
                  ["1","2","3","+"
                  ,"4","5","6","-"
                  ,"7","8","9","*"
                  ,"0","c","=","/"]                ]
tableAttachDefaults table entry 0 5 0 1
sequence_ [on b buttonActivated $ callback b | b <- buttons]
on window objectDestroy mainQuit
widgetShowAll window
mainGUI
```



## Threepenny-GUI

- Bibliothek `Threepenny-GUI` (`https://wiki.haskell.org/Threepenny-gui`)
- einfache, sehr stabile und Plattform-unbhängige Lösung für eine GUI.
- Benutzt den Browser als Display.
- Sie kompiliert nach JavaScript.
- Ausführung des Programms erzeugt einen Webserver der GUI unter `http://localhost:8023` bereitstellt
- Programmierung: siehe Doku, wir zeigen Taschenrechner






## Taschenrechner mit Threepenny-GUI

```
import Control.Monad
import Data.IORef
import Data.Char
import qualified Graphics.UI.Threepenny as UI
import Graphics.UI.Threepenny.Core
main :: IO ()
main = startGUI defaultConfig setup
setup :: Window -> UI ()
setup w = void $ do
 entryItem <- UI.input # set UI.value "0"
 let start     = (id, 0.0)
 cSt <- liftIO $ newIORef start
 let
  callback b (l:_) =  do
   calcStep l
   x <- liftIO $ readIORef cSt
   element entryItem # set UI.value (show $ snd x)
  calcStep x | isDigit x
    = digit (fromIntegral $ digitToInt x)
  calcStep o | o `elem` "+-*/" = oper o
  calcStep '=' = total
  calcStep 'c' = clear
  calcStep  _  = return ()
  oper op = liftIO $ modifyIORef cSt (\(f,n)->(op (f n),0))
  total = liftIO $ modifyIORef cSt (\(f,n)->(id,f n))
  clear = liftIO $ modifyIORef cSt (\(f,n)->if n == 0.0
                         then start else (f,0.0))
  digit i = liftIO $ modifyIORef cSt (\(f,n)->(f,n*10+(fromIntegral i)))
  ...
```

```
...
return w # set title "Buttons"
buttons <- replicateM 16 UI.button
let labeledbuttons = zip buttons ["1","2","3","+"
                                 ,"4","5","6","-"
                                 ,"7","8","9","*"
                                 ,"0","c","=","/"]
sequence_ [element b # set UI.text t | (b,t) <- labeledbuttons]
sequence_ [on UI.click b  (\_ -> callback b t)
                | (b,t) <- labeledbuttons     ]
getBody w #+
  [UI.table #+
    (   [UI.tr #+ [(UI.td # set UI.colspan 4)
                      #+ [element entryItem]  ]  ]
     ++ [UI.tr #+ [UI.td #+ [element b]
                  | b <- take 4 buttons]]
     ++ [UI.tr #+ [UI.td #+ [element b]
                  | b <- take 4 $ drop 4 buttons]]
     ++ [UI.tr #+ [UI.td #+ [element b]
                  | b <- take 4 $ drop 8 buttons]]
     ++ [UI.tr #+ [UI.td #+ [element b]
                  | b <- take 4 $ drop 12 buttons]]
    )
  ]
```