# GUI Applikationen mit GTK Haskell

Sven Eric Panitz

2. Juni 2022

## Inhaltsverzeichnis



# 1 GTK

In diesem Lehrbrief verwenden wir die Bibliothek Gtk in der Version 3. Gtk ist eine C Bibliothek zum Schreiben von GUI-Anwendungen. Sie entstammt ursprünglich aus dem Projekt der Bildbearbeitungssoftware Gimp und wurde lange *GTK+* bezeichnet, verzichtet aber mitlerweile auf das `+` im Namen.

Es gibt Anbindungen der GTK Bibliothek an viele verschiedene Programmiersprachen. So gibt es auch eine Anbindung an Haskell. Es gibt sogar zwei Anbindungen an Haskell: die Bibliothek Gtk2Hs und haskell-gi. Wir betrachten hier Gtk2hs.

Um in Haskell GTK-Anwendungen zu schreiben, wird zunächst die GTK Bibliothek und alles darin verwendeten C Bibliotheken für Entwickler benötigt und schließlich die Gtk2hs Bibliotek, die sich mit `cabal` installieren lässt.

Dann kann man beginnen, erste kleine GUI-Biblioztheken zu entwickeln.

Haskell: Calculator

```
> module Calculator where

> import Graphics.UI.Gtk
> import Control.Concurrent
> import Control.Monad.IO.Class
> import Data.Char
> import Data.List
```

Wir beginnen mit dem wahrscheinlich kürzesten GUI-Programm, das ein leeres Fenster öffnet.

Alle GTK-Funktion in Haskell laufen in der IO-Monade. Das GTK Gui ist zu initialisieren mit `initGUI`. Dann können Gui-Komponenten erzeugt werden und abschließend mit `mainGUI` der Hauptsteuerfaden für die GUIs gestartet werden.

Haskell: Calculator

```
> moin1 = do
>   initGUI
>   window <- windowNew
>   widgetShowAll window
>   mainGUI
```

Die verschiedenen GUI-Komponenten haben Konstruktorfunktionen, die mit dem Namen der Komponente beginnen und dem Wort `New` enden. So gibt es eine Funktion `windowNew`, zum Erzeugen eines neuen Fensters.

Das Programm `moin1` öffnet ein erstes Fenster.

Als nächstes wollen wir dem Fenter Eigenschaften geben. Hierzu gibt es eine allgemeine Funktion `set`, die eine Liste von Attribut-Wert-Paare erhält. Diese sind durch den Infixkonstrukor `:=` getrennt.

Einem Fenster lassen sich so zum Beispiel ein Rand und ein Titel geben.

Haskell: Calculator

```
> moin2 = do
>   initGUI
>   window <- windowNew
>   set window [ containerBorderWidth := 10
>              , windowTitle := "Hier könnte Ihre Werbung stehen"]
```

Um einer Komponente eine Ereignissbhandlung für bestimmte Ereignisse zu geben, wird die Funktion `on` verwendet, die oft in Infixnotation notiert wird. Wir geben dem Fenster die Eigenschaft, dass ein Schließen des Fensters (`deleteEvent`) dazu führt, dass das ganze Programm beendet wird.

Haskell: Calculator

```
>   window `on` deleteEvent $ do liftIO mainQuit
>                                return False
>   widgetShowAll window
>   mainGUI
```

Als nächstes betrachten wir, wie eine Komponente Unterkomponente erhält und wie diese positioniert werden können. Hierzu soll unsere Anwendung zwei Knöpfe und ein Anzeigedisplay bekommen. Die Knöpfe sind vom Typ `Button` und haben eine Konstruktorfunktion `buttonNew`, ein Textfeld ist vom Typ `Entry` und hat die Konstruktorfunktion `entryNew`.

Zum Gruppieren gibt es die Komponenten `HBox` und `VBox` oder komplexere Komponenten wie `Grid`. Wir begnügen uns damit, die drei inneren Kompnenten horiziontal anzuordnen und nehmen ein Objekt des Typs `HBox`. Dieser fügen wir mit der Funktion `boxPackStart` die einzelnen Subkomponenten hinzu.

Haskell: Calculator

```
> moin3 = do
>   initGUI
>   window <- windowNew
>   set window [ containerBorderWidth := 10]
>   box <- hBoxNew True 1
>   b1 <- buttonNew
>   set b1 [ buttonLabel := "Knopf 1" ]
>   b2 <- buttonNew
>   set b2 [ buttonLabel := "Knopf 2" ]
>   display <- entryNew
>   boxPackStart box b1 PackGrow 0
>   boxPackStart box b2 PackGrow 0
>   boxPackStart box display PackGrow 0
>   set window [ containerChild := box]
>   window `on` deleteEvent $ do liftIO mainQuit
>                                return False
>   widgetShowAll window
>   mainGUI
```

Damit wir eine sinnvolle erste GUI-Anwendung bekommen, erhält das GUI jetzt noch ein simples Modell, auf dem es agiert. Das Modell soll nur aus einer Zahl bestehen, die angezeigt und mit den Knöpfen hoch oder herunter gezählt werden kann.

Für das Modell wird eine MVar erzeugt, auf die alle GUI-Komponenten zugreifen.

Haskell: Calculator

```
> moin4 = do
>   initGUI
>   window <- windowNew
>   mv <- newMVar 0
>   box <- hBoxNew True 1
>   b1 <- buttonNew
>   set b1 [ buttonLabel := "+" ]
>   b2 <- buttonNew
>   set b2 [ buttonLabel := "-" ]
>   display <- entryNew
>   boxPackStart box b1 PackGrow 0
>   boxPackStart box b2 PackGrow 0
>   boxPackStart box display PackGrow 0
```

Jetzt geben wir den Knöpfen mit der Funktion `on` jeweils eine Ereignisbehandlung. In dieser wird der Wert in der MVar modifiziert. Anschließend wird der MVar-Wert, also das Datenmodell, gelesen und in der Textkomponente als Text gesetzt.

Haskell: Calculator

```
>   b1 `on`buttonActivated $
>         do
>            modifyMVar_ mv (\v->return (v+1))
>            v <- readMVar mv
>            set display [ entryText := show v ]
>
>   b2 `on`buttonActivated $
>         do
>            modifyMVar_ mv (\v->return (v-1))
>            v <- readMVar mv
>            set display [ entryText := show v ]
>
>   set window [ containerChild := box]
>   window `on` deleteEvent $ do liftIO mainQuit
>                                return False
>   widgetShowAll window
>   mainGUI
```

Im Prinzip ist das alles, was man wissen muss, wie es funktioniert. Nun geht es wie bei den meisten GUI-Bibliotheken hauptsächlich darum, die Dokumentation der vielen Komponenten zu lesen und die vielen Attribute, die gesetzt werden können, zu kennen.

# 2 Eine Taschenrechneranwendung

Exemplarisch soll in diesem Kapitel ein kleiner Taschenrechner für die Grundrechenarten entwickelt werden. Dabi wollen wir versuchen die übliche Trennung der Architektur einzuhalten, die eine GUI-Anwendung in der Regel hat.

## 2.1 Modell

Wir beginnen mit einem Modell. Ein Taschenrechner braucht ein Modell für zwei Zahlenregister und einen Operator auf den Zahlen in den Registern. In einem Taschenrechner wird der Operator zumeist als Infixoperation eingegeben. So haben wir eine Datentyp als Modell, der diese drei Argumente enthält.

Haskell: Calculator

```
> data State
>  = State{i1::Integer, i2:: Integer, op:: (Integer->Integer->Integer)}
```

Wir defieren eine Funktion, die den neuen Zustand nach Eingabe einer weiteren Ziffer definiert. Die zusätzliche eingegebene Ziffer bezieht sich auf die erste der beiden Zahlen im Modell.

Haskell: Calculator

```
> addDigit d st@(State i1 i2 op)
>  |d>=0 && d<10 =  State (10*i1+d) i2 op
>  |otherwise = st
```

Wenn ein Operator eingegeben wird, dann werden die beiden gespeicherten Zahlen mit dem vorherigen Operator verrechnet und der neue Operator vermerkt für die nächste Rechnung. Das eine Rechenregister wird auf 0 gesetzt für die Eingabe eines neuen Operanden.

Haskell: Calculator

```
> addOp op (State i1 i2 op1) = State 0 (op1 i2 i1) op
```

Im initalen Zustand sind beide rechenregister auf 0 gesetzt.

Haskell: Calculator

```
> initStat = State 0 0 (\x y -> y)
```

Folgende 6 Operatoren sollen erlaubt sein:

Haskell: Calculator

```
> ops = "+-*/%="
```

Die Operatoren werden mit einer zweistellige Funktion auf Integer assoziiert.

Haskell: Calculator

```
> readOP "+" = (+)
> readOP "-" = (-)
> readOP "*" = (*)
> readOP "/" = div
> readOP "%" = mod
> readOP "=" = \x y-> x
> readOP _ = \x y-> y
```

Bevor wir im nächsten Abschnitt eine GUI-Anwendung für den Taschenrechner schreiben, sei eine interaktive Kommandozeilenanwendung definiert. Die späteren Knopfeingabe auf dem Taschenrechner-GUI sind hier Tastatureingaben, die mit `getChar` erfragt werden:

Haskell: Calculator

```
> moin5 s = do
>  c<-getChar
>  putStr "\n"
>  let s'@(State r1 r2 op) =
>        if isDigit c
>        then addDigit (toInteger (ord c-ord '0')) s
>        else if elem c ops
>        then addOp (readOP [c]) s
>        else s
>  print (if elem c ops then r2 else r1)
>  moin5 s'
```

## 2.2 Gui Komponenten

Jetzt soll für das Datenmodell ein kleines GUI entwickelt werden. Hierzu empfiehlt es sich, zunächst einmal alle relevanten GUI-Komponenten in einem Typ zu sammeln. Wir brauchen Knöpfe für die Ziffern, Knöpfe für die Operatoren und eine Anzeige.

Haskell: Calculator

```
> data GuiControls
>  = GuiControls
>    { digitButtons     :: [Button]
>    , display          :: Entry
>    , operatorButtons  :: [Button]
>    }
```

Die Komponenten sind alle zu erzeugen. Wir erzeugen die Liste der 10 Zifferntasten, die Liste für die 6 Opoeratortasten und das Objekt für die Anzeige. Alle werden im Datenobjekt vom Typ `GuiControls` zusammen gefasst.

Haskell: Calculator

```
> digits = [1..9]++[0]
>
> mkGuiControls = do
>   ds <- sequence (fmap (mkBtn.show) digits)
>   display <- entryNew
>   set display [ entryEditable := False
>               , entryXalign   := 1
>               , entryText     := "0" ]
>   os <- sequence$fmap mkBtn$fmap (\x->[x]) ops
>   return (GuiControls ds display os)
```

Als Hilfsfunktion diente hierbei die folgende Funktion, zur Erzeugung eines Knopfes mit einer Knopfbeschriftung.

Haskell: Calculator

```
> mkBtn label = do
>  btn <- buttonNew
>  set btn [ buttonLabel := label ]
>  return btn
```

## 2.3 Layout

Als nächsten Schritt sollen die Komponenten zueinander angeordnet werden. Üblicher Weise sind die Tasten eines Taschenrechners gleich groß in Tabellenform angelegt. Hierzu gibt es in Gtk die Komponenten vom Typ `Grid`.

Haskell: Calculator

```
> layoutGui gui = do
>   grid <- gridNew
>   gridSetRowHomogeneous grid True
```

Die Anzeige wird über die ganze obere Zeile des Grid gesetzt. Die Funktion `gridAttach` ermöglicht es nicht nur eine Komponente per Spalten- und Zeilenangabe zu platzieren, sondern auch über mehrer Zeilen und Spalten zu strecken.

Haskell: Calculator

```
>   gridAttach grid (display gui) 0 0 4 1
```

Die Zifferntasten werden auf den ersten drei Spalten ab der zweiten Zeile gruppiert.

Haskell: Calculator

```
>   sequence $ fmap (\(p,b)->gridAttach grid b (p`mod`3) (1+p`div`3) 1 1)
>            $ zip [0..]
>            $ digitButtons gui
```

Jetzt bleiben in der rechtesten Spalte drei Felder und in der untersten Zeile drei Felder. Wir teilen die Operatortasten in zwei Gruppen:

Haskell: Calculator

```
>   let (op1,op2) = splitAt 3 $ operatorButtons gui
```

Die ersten drei kommen in die letzte Spalte;

Haskell: Calculator

```
>   sequence $ fmap (\(p,b)->gridAttach grid b 3 p 1 1 )
>            $ zip [1..] op1
```

Die übrigen drei in die letzte Zeile:

Haskell: Calculator

```
>   sequence $ fmap (\(p,b)->gridAttach grid b p 4 1 1 )
>            $ zip [1..] op2
```

Das Layout ist fertig gesetzt und wird als neue Komponente zurück gegeben.

Haskell: Calculator

```
>   return grid
```

## 2.4 Aktionsbehandlung

Das fertige GUI, der View, ist jetzt mit dem Modell zu verknüpfen, indem die GUI-Ereignisses behandelt werden und der jeweiligen Zustand des Modells angezeigt wird.

Hierzu ist zunächst eine MVar mit dem initialen Zustand anzulegen.

Haskell: Calculator

```
> addGuiEvents gui = do
>  st <- newMVar initStat
```

Auf diese werden alle Ereignisse Bezug nehmen. Zunächst kümmern wir uns um die Knöpfe für die Ziffern. Diese modifizieren die Zustandsvariable. Anschließend lesen sie aus dem Zustand den Wert des ersten Registers, um diesen in der Anzeige darzustellen.

Haskell: Calculator

```
>  sequence
>     [ on b buttonActivated$
>           do
>              modifyMVar_ st (return . addDigit i)
>              s<-readMVar st
>              set (display gui) [entryText := show (i1 s)]
>    | (i,b) <- zip digits (digitButtons gui)]
```

Ähnlich läuft es mit den Tasten für die Operatoren. Die Operation wird aus dem Label gelesen und mit dieser der Zustand modifiziert. Jetzt wird das zweite Rechenregister angezeigt, in dem nun das Ergebnis der letzten Operation liegt.

Haskell: Calculator

```
>  sequence
>     [ on b buttonActivated$
>         do
>            modifyMVar_ st
>              (\s->buttonGetLabel b>>= \l -> return$ addOp (readOP l) s)
>            s <- readMVar st
>            set (display gui) [ entryText := show (i2 s) ]
>     | b<-operatorButtons gui]
>
>  return gui
```

Alles zusammen ergibt die neue GUI-Komponenten eines Taschenrechners.

Haskell: Calculator

```
> calculatorNew = do
>   gui <- mkGuiControls
>   addGuiEvents gui
>   grid <- layoutGui gui
>   return grid
```

### 2.5 Hauptprogramm

Das Hauptprogramm verwendet jetzt den Taschenrechner wie jede andere GUI-Komponenten, indem er erzeugt und einem Fenster hinzugefügt wird.

Haskell: Calculator

```
> moin = do
>   initGUI
>   window <- windowNew
>   set window [ containerBorderWidth := 10]
>   calculator <- calculatorNew
>   set window [ containerChild := calculator]
>   window `on` deleteEvent $ do liftIO mainQuit
>                                return False
>   widgetShowAll window
>   mainGUI
```

Ich denke, es ist auch in Haskell sinnvoll, zunächst Modell und Ansicht zu trennen und dann über die Events mit einer MVar zu verbinden, um softwaretechnologisch leicht wartbare getrennte Komponenten zu bekommen.

## 3 Aufgaben

**Aufgabe 1** Nutzen Sie die Möglichkeit mit Gtk GUIs zu programmieren, indem Sie ein Programm nehmen, das bisher vielleicht nur eine Kommandozeileninteraktion zulässt, um es mit einer grafischen Benutzeroberfläche zu versehen, mit der die einzelnen Funktionen aufgerufen und das Ergebnis dieser angezeigt werden kann. Das können zum Beispiel die Programme zu Sudoku, Kreuzworträtsel, Audioverarbeitung, RSA oder Strategiespielen aus den anderen Lehrbriefen sein.

## 4 Lernzuwachs

- GUI Programmierung

- die IO Monade im Einsatz
- Einsatz von MVar