

FUNKTIONALE PROGRAMMIERUNG

GRAPHISCHE BENUTZEROBERFLÄCHEN MIT GTK2HS

Andreas Abel, Steffen Jost

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

14. Mai 2012

HINTERGRUND-INFOS ZU GTK

- GTK ist Abkürzung für *The GIMP ToolKit*
- Portable Schnittstelle für Graphische Benutzeroberflächen (GUIs).
- Komponenten:
 - ① GLib (Kernbibliothek): Kompatibilitätsschicht, Ereignisschleife, Threads, dynamisches Laden, Objektsystem.
 - ② Pango: Textdarstellung, Internationalisierung.
 - ③ Cairo: 2D Graphik auf verschiedenen Medien (Dateien, Windows...).
 - ④ ATK: Accessibility.

IMPERATIVE GUI-PROGRAMMIERUNG

- Benutzerinteraktion ist I/O.
- Eingabe: Ereignisse (Clicks, Tastendrucke, interne Nachrichten, Nachrichten von anderen Applikationen).
- Ausgabe: Graphische Darstellung von Inhalt und Steuerung.
- GUI-Programmierung ist imperativ, lebt in der *IO*-Monade.
- Ereignisse werden durch *callbacks* behandelt.
- In Haskell elegant gelöst mittels verzögerter Auswertung (lazyness).
- Problem: Ereignisbehandler müssen Zustand der Applikation ändern.

HELLO WORLD!

```
import Graphics.UI.Gtk
main :: IO ()
main = do
  initGUI
  window ← windowNew
  label   ← labelNew $ Just "Hello world!"
  set window [windowDefaultWidth := 200,
              windowDefaultHeight := 200,
              containerChild      := label]
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI
```

INITIALISIERUNG UND EREIGNISSCHLEIFE

```
module Graphics.UI.Gtk.General.General where
```

```
initGUI  :: IO [String]
```

```
-- Initialize GUI toolkit and parse Gtk specific arguments.
```

```
-- The remaining arguments are returned.
```

```
mainGUI :: IO () -- Run the Gtk+ main event loop.
```

```
mainQuit :: IO () -- Exit the main event loop.
```

Widget

Blend of window and gadget, coined by George S. Kaufman in his play Beggar on Horseback (1924).

- 1 A small scraping tool consisting of a blade and a handle, commonly used to remove paint from glass and other smooth surfaces. (Spachtel)
- 2 A floating widget or other device inside a beer can, meant to create foam when opened.
- 3 An unnamed, unspecified, or hypothetical manufactured good or product. (Dingsbums, Vorrichtung)
- 4 An object of fiction or obfuscation (Gigawidget).
- 5 A component of a graphical user interface that the user interacts with.

ATTRIBUTE LESEN UND SCHREIBEN

```
module System.Glib.Attributes where  
data ReadWriteAttr o a b  
data AttrOp o = forall a b.(:=) (ReadWriteAttr o a b) b  
                | ...  
get :: o → ReadWriteAttr o a b → IO a  
set :: o → [AttrOp o] → IO ()
```

Beispiele:

```
value ← get button buttonLabel  
set button [buttonLabel := value, buttonFocusOnClick := False]
```

AttrOp o ist ein existentieller Datentyp. Für einen Bewohner *attr := value :: AttrOp o* gibt es zwei Typen *a, b*, so dass *attr :: ReadWriteAttr o a b* und *value :: b*. Das *forall* kommt von *(:=) :: forall a b. ReadWriteAttr o a b → b → AttrOp o*.

module *System.Glib.Signals* **where**

data *ConnectId* *o*

- If you ever need to disconnect a signal handler then you will
- need to retain the *ConnectId* you got when you registered it.

disconnect :: *GObjectClass obj* \Rightarrow *ConnectId obj* \rightarrow *IO ()*

module *Graphics.UI.Gtk.Abstract.Widget* **where**

widgetShowAll :: *WidgetClass self* \Rightarrow *self* \rightarrow *IO ()*

- Recursively shows a widget, and any child widgets.

onDestroy :: *WidgetClass w* \Rightarrow *w* \rightarrow *IO ()* \rightarrow *IO (ConnectId w)*

PACKEN UND QUETSCHEN

Horizontal packen.

```
module Graphics.UI.Gtk.Layout.HBox where  
hBoxNew :: Bool -- True if all children given equal space.  
  →      Int    -- number of pixels between children.  
  →      IO HBox  
module Graphics.UI.Gtk.Abstract.Box where  
data Packing = PackGrow -- widget grows with box: text area  
           | PackRepel  -- only padding grows: dialog box  
           | PackNatural -- stay where you are: menu bar  
boxPackStart :: (BoxClass self, WidgetClass child) ⇒  
               self → child →  
               Packing → Int → IO ()  
-- Int is extra padding
```

Vertikal packen analog mit `vBoxNew`.

HORIZONTALE PACKUNG AM BEISPIEL

```
main = do
  initGUI
  window ← windowNew
  hbox   ← hBoxNew True 10
  button1 ← buttonNewWithLabel "Button 1"
  button2 ← buttonNewWithLabel "Button 2"
  set window [windowDefaultWidth := 200,
              windowDefaultHeight := 200,
              containerBorderWidth := 10,
              containerChild      := hbox]
  boxPackStart hbox button1 PackGrow 0
  boxPackStart hbox button2 PackGrow 0
  ...
```

SCHACHBRETTER UND TABELLENLAYOUTS

```
module Graphics.UI.Gtk.Layout.Table where  
tableNew :: Int      -- number of rows  
         → Int      -- number of columns  
         → Bool     -- homogeneous size?  
         → IO Table  
tableAttachDefaults :: (TableClass self, WidgetClass widget)  
                    ⇒ self → widget -- table, child  
                    → Int → Int    -- left and right column to attach  
                    → Int → Int    -- top and bottom row to attach  
                    → IO ()
```

BEISPIEL FÜR TABELLENLAYOUT

createButton :: *Table* → (*Int*, *Int*) → *IO Button*

createButton table (x, y) = **do**

b ← *buttonNew*

onClicked *b* (*buttonPress* (x, y))

tableAttachDefaults table *b* x (x + 1) y (y + 1)

return *b*

main = **do** ...

table ← *tableNew* *ysize* *xsize* *True*

let *rows* = [[(x, y) | x ← [0..*xsize* - 1]] | y ← [0..*ysize* - 1]]

buttons ← *mapM* (*mapM* (*createButton* table)) *rows*

 ...

mapM :: *Monad m* ⇒ (*a* → *m b*) → [*a*] → *m [b]*

-- from Prelude or Control.Monad

“GLOBALE” VARIABLEN UND IOREF

Simulation globaler Variablen mittels Record, z.B.

```
data GlobVar = GlobVar { ro1 :: Int
                        , ro2 :: [(Int, Int)]
                        , rw1 :: IORef Int
                        , rw2 :: IORef [IORef [Int]] }
```

```
func1 :: GlobVar → Int → IO [Int]
```

```
func2 :: GlobVar → a → IO a
```

Jede Funktion erhält *GlobVar* als Parameter. Veränderbare Var.:

```
module Data.IORef where
```

```
data IORef a
```

```
newIORef :: a → IO (IORef a)
```

```
readIORef :: IORef a → IO a
```

```
writelIORef :: IORef a → a → IO ()
```

BEISPIELAPPLIKATION: TIC TAC TOE

- Erstelle 9 Buttons, ohne Label.
- Klick auf Button ist Zug des menschlichen Spielers.
- Ereignisbehandlung setzt Label und berechnet Antwort des Computerspielers.
- Muss auch Zustand des Buttons der Antwort ändern.
- Ereignisbehandlung eines Buttons braucht Zugriff auf alle Buttons.
- Lösung: Erstelle zuerst Buttons, installiere dann Ereignisbehandlung für alle.
- Kann man die Ereignisbehandlung auch bei der Erstellung einhängen?

BEISPIELAPPLIKATION: TIC TAC TOE

- Erstelle 9 Buttons, ohne Label.
- Klick auf Button ist Zug des menschlichen Spielers.
- Ereignisbehandlung setzt Label und berechnet Antwort des Computerspielers.
- Muss auch Zustand des Buttons der Antwort ändern.
- Ereignisbehandlung eines Buttons braucht Zugriff auf alle Buttons.
- Lösung: Erstelle zuerst Buttons, installiere dann Ereignisbehandlung für alle.
- Kann man die Ereignisbehandlung auch bei der Erstellung einhängen?
- Idee: Zirkuläres Programm.

VERSUCH EINER ZIRKULÄREN KONSTRUKTION

```
data State -- Tic Tac Toe board
data App = App { stateRef :: IORef State
                 , buttons :: [[Button]] } -- global vars of appl.
buttonPress :: App → (Int, Int) → IO () -- button event handler
createButton :: App → Table → (Int, Int) → IO Button
createButton app table (x, y) = do
  b ← buttonNew
  onClicked b (buttonPress app (x, y))
  tableAttachDefaults table b x (x + 1) y (y + 1)
  return b
let app = App {
  stateRef = newIORef initialState,
  buttons = map (map (createButton app table)) rows }
-- DOES NOT WORK: createButton is monadic!
```

ZIRKULÄRE MONADISCHE PROGRAMME

- Monadische Rekursion:

```
module Control.Monad.Fix where  
class Monad m  $\Rightarrow$  MonadFix m where  
    mfix :: (a  $\rightarrow$  m a)  $\rightarrow$  m a  
instance MonadFix IO
```

- Den Knoten zuziehen (tying the knot):

```
do app  $\leftarrow$  mfix $  $\lambda$ app'  $\rightarrow$   
    do bs  $\leftarrow$  mapM (mapM (createButton app' table)) rows  
        return $ App { stateRef = ...  
                      , buttons = bs  
                    }
```

RECURSIVES **do**

```
do app ← mfix $ λapp' →  
  do bs ← mapM (mapM (createButton app' table)) rows  
  return $ App { stateRef = ...  
                , buttons = bs  
                }
```

Mit GHC-Spracherweiterung (Flag -XDoRec):

```
{-# LANGUAGE DoRec #-}  
do rec bs      ← mapM (mapM (createButton app table)) rows  
  let app = App { stateRef = ...  
                , buttons = bs  
                }
```

MENÜ UND WERKZEUGLEISTE

- Menü, Werkzeugleisten und Tastenkombinationen starten *Actions*.
- Aktionen können aktiviert und deaktiviert werden.

actionNew ::

String -- name : unique name for the action
→ *String* -- label : displayed in menu items and on buttons
→ *Maybe String* -- tooltip
→ *Maybe String* -- stockId : icon to be displayed
→ *IO Action*

onActionActivate :: *ActionClass self* ⇒ *self* → *IO ()* →
 IO (ConnectId self)

actionSetVisible :: *ActionClass self* ⇒ *self* → *Bool* → *IO ()*

actionSetSensitive :: *ActionClass self* ⇒ *self* → *Bool* → *IO ()*

XML MENÜBESCHREIBUNG

```
uiDecl = ""  
"<ui>"  
"  <menubar>"  
"    <menu action=\"FILE_MENU\">"  
"      <menuitem action=\"QUIT\"/>"  
"    </menu>"  
"  </menubar>"  
"  <toolbar>"  
"    <toolitem action=\"QUIT\"/>"  
"  </toolbar>"  
"</ui>"
```

```

createMenu :: VBox → IO ()
createMenu box = do
  actFileMenu ← actionNew "FILE_MENU" "File" Nothing Nothing
  actQuit ← actionNew "QUIT" "Quit"
    (Just "Exit Tic Tac Toe") (Just stockQuit)
  onActionActivate actQuit mainQuit
  actGroup ← actionGroupNew "ACTION_GROUP"
  mapM (actionGroupAddAction actGroup) [actFileMenu, actQuit]
  ui ← uiManagerNew
  uiManagerAddUiFromString ui uiDecl
  uiManagerInsertActionGroup ui actGroup 0
  Just menubar ← uiManagerGetWidget ui "/ui/menubar"
  boxPackStart box menubar PackNatural 0
  Just toolbar ← uiManagerGetWidget ui "/ui/toolbar"
  boxPackStart box toolbar PackNatural 0

```

RESSOURCEN

- GTK+ Projekt: <http://www.gtk.org/>
- gtk2hs Homepage: <http://haskell.org/gtk2hs/>
- gtk2hs Tutorial:
<http://home.telfort.nl/sp969709/gtk2hs/>
- Sprung in Referenz:
<http://hackage.haskell.org/package/gtk-0.12.3>
- Installation:
<http://www.haskell.org/haskellwiki/Gtk2Hs>