Graphical User Interfaces in Haskell

Paul Martinez – CS 240H – Stanford University

June 11, 2014

Introduction

Anyone who wishes to build an entire application in Haskell may eventually need to build a Graphical User Interface (GUI). While Haskell does many things well, it does not have a standard GUI library and, as the Haskell website describes the situation, existing GUI libraries are "more or less incomplete." We will examine two libraries used to build GUIs in Haskell in this paper.

The first library is Threepenny-gui. Threepenny-gui takes an interesting approach of taking advantage of a user's already installed browser to display information using an interesting sort of client-server setup. The second library is Gtk2Hs which is built on top of the already-existing Gtk+ toolkit, a widely used GUI library that supports multiple platforms.

Threepenny-gui

Threepenny-gui takes an interesting approach to user-interfaces by taking advantage of a user's browser to provide all the UI primitives and display content to the user. When using Threepenny one uses Haskell to construct a webpage in HTML and specify how various elements should interact. Threepenny then converts these specifications into Javascript code that is executed on a webpage. Running a Threepenny application starts a server on localhost that can then be visited on your normal web browser. While you interact with the program via the browser, the program can still interact with the server, so you still have all the power of a regular native program at your control.

Examples

Here we give a simple example of a traditional Hello World program using Threepenny:

```
import Graphics.UI.Threepenny

main :: IO ()
main = do
  startGUI defaultConfig sayHello

sayHello :: Window -> UI ()
sayHello window = do
  getBody window #+ [string "Hello, world!"]
  return ()
```

A brief overview of the keypoints here: `startGUI` is the function that starts the Threepenny server. It has type `Config -> (Window -> UI ()) -> IO ()`. It accepts a configuration, which allows specifying things such as port number and the location of resources, and a function that takes a `Window` object and performs a `UI` action. A default configuration object is provided. The function is a setup function that gets called everytime a connection is opened to the Threepenny server. The `Window` object represents the window DOM element of the newly created webpage.

Our setup function here is gets the body DOM element from the window and appends (`#+`) a string containing the text "Hello, world!". Everything is done in a special UI Monad which is a small wrapper around the IO Monad but keeps track of some additional information. It is an instance of `MonadIO`, so to list all the files in the same directory as your program, you could replace the `sayHello` function above with:

```
listFiles :: Window -> UI ()
listFiles window = do
  files <- liftIO $ getDirectoryContents "."
  getBody window #+ (map toDiv files)
  return ()

toDiv :: String -> UI Element
toDiv str = UI.div #+ [string str]
```

This shows how one can easily take advantage of system level features in Threepenny. As a final example, we will show how a simple example of how to respond to user input. We replace the `listFiles` function above with:

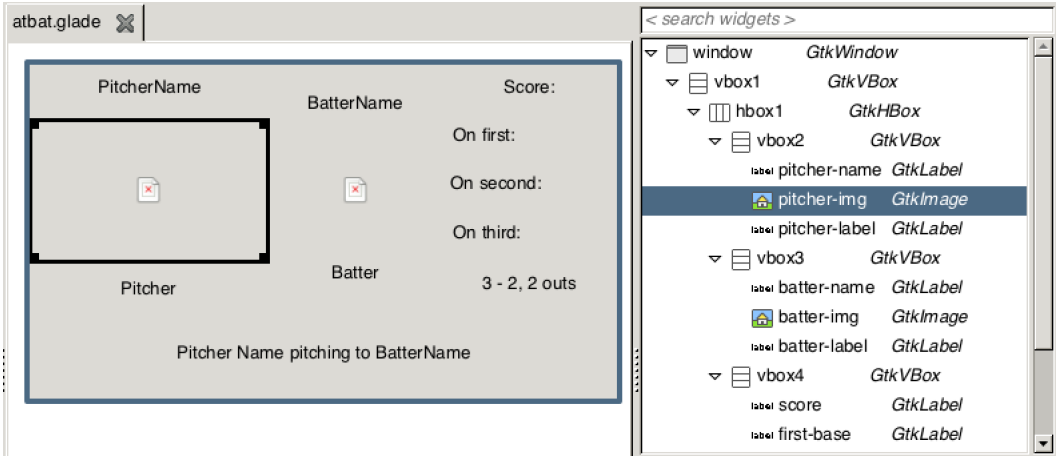
```
clickMe :: Window -> UI ()
clickMe window = do
  button <- UI.button #+ [string "Click me!"]
  getBody window #+ [return button]

  on click button $ \_ -> do
    getBody window #+ [toDiv "You clicked the button!"]
```

This function simply prints "You clicked the button" to the window whenever the button is clicked.

Overall Threepenny has some interesting things to offer. By taking advantage of the DOM and CSS, one can create very appealing and reactive UIs. Being able to access the filesystem and other system level things provides many more possibilities than a simple web app. Some downsides could be that, because everything gets compiled down to Javascript, debugging is more difficult, and in the end one may feel that the entire system is trying to hard to force Haskell into a situation where it's necessarily designed to excel. (Threepenny does provide a way of calling Javascript directly). Additionally, since you are working within the browser you may have less control than you would like. (You can't create real popup windows or control window positioning for example.) But being able to use native browser UI primitives allows one to build modern looking applications very quickly.

Gtk2Hs

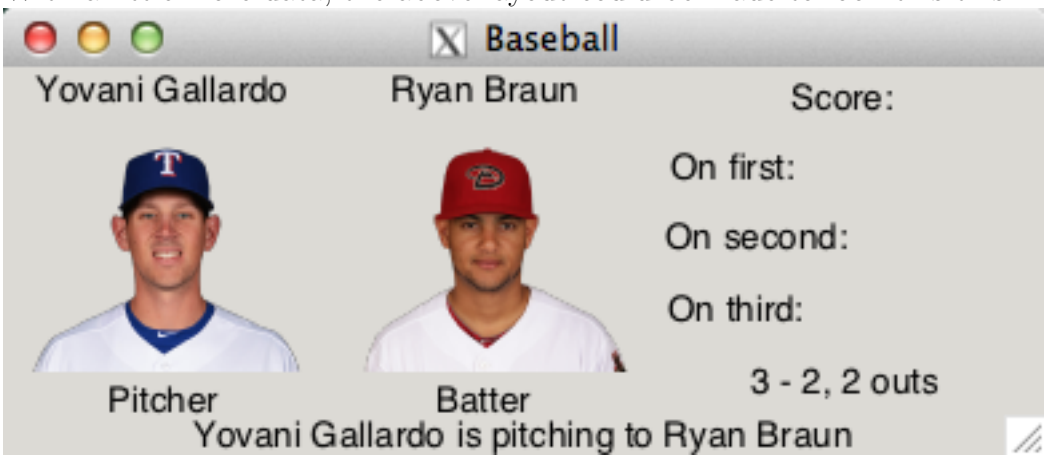
Gtk2Hs is a library built on top of the multiplatform GUI library Gtk+. It also uses Glade, a user interface designer that saves layout as XML files. Below is example of using Glade to build a layout for a UI that displays current information about a baseball game.



Using Glade makes arranging layouts very easy and it's also very easy to populate them with data. The following example shows how one could set the name of a player and display their picture for the above layout.

```
main :: IO ()
  initGUI
  Just xml <- xmlNew "layout.glade"
  pitcher <- xmlGetWidget xml castToLabel "pitcher-name"
  set pitcher [labelText := "Yovani Gallardo"]
  pitcherImg <- xmlGetWidget xml castToImage "pitcherImg"
  imageSetFromFile img "yovani.png"
  window <- xmlGetWidget xml castTowindow "window"
  widgetShowAll window
  mainGUI
```

With a little more data, the above layout could be made to look this this:



The important thing to do in with a GUI library is listen to and react to events. The following snippet shows how to create a button and a label that will display how many seconds it has been since the button was clicked.

```
1  timeLabel <- labelNew Nothing
2  startButton <- buttonNewWithLabel "Exit"
3
4  startButton `on` buttonActivated $ do
5    forkIO $ do
6      let printTime t = do
7        threadDelay 1000000
8        postGUIAsync $ labelSetText timeLabel (show t)
9        printTime (t + 1)
10     printTime 1
11   return ()
```

I created a small program using Gtk2Hs that fetches live data for baseball games from ESPN's GameCast then displays the information in popup window. It fetches the data every 30 seconds and displays the current state of the game for seven seconds. The code for the program can be viewed here: `http://github.com/PaulJuliusMartinez/baseball`.