An Introduction to Gtk2Hs, a Haskell GUI Library

by Kenneth Hoste ⟨kenneth.hoste@UGent.be⟩

This article is an introduction to Gtk2Hs, one of many Haskell GUI libraries. We introduce the library by means of a small example, which we will build from scratch during this article. The emphasis is on the Gtk2Hs related code.

Introduction

Since this article is an introduction to **Gtk2Hs**, we will start with some words on the library itself: the structure of the library, its advantages and disadvantages, and how to install and use it.

In order to make the explanation of the basic principles behind Gtk2Hs a little bit easier, we use an example application: the Memory game. The code needed to run the application was written to serve as example code for this article, and thus is not meant to be fully working, or even bug-free.

Because one of the big advantages of the Gtk2Hs library is the support of Glade, we show how we can access a GUI description created with Glade. The main part discusses the Gtk2Hs related functionality of the Memory application. This includes setting up communication between the different parts of the program. We show how we can ensure the interaction with the user runs smoothly.

To conclude the article, we review the important items we discussed, and try to see what the future could bring us.

What is Gtk2Hs?

As we mentioned above, Gtk2Hs is a **GUI library for Haskell**. It is based on **Gtk+** (version 2.6), a multi-platform toolkit for creating graphical user interfaces. Some of its features we use in this article include nearly complete coverage of the Gtk+ toolkit, API documentation (still in development), bindings for several Gnome modules (more specifically: libglade for loading GUIs from xml files at runtime, GConf for storing application preferences and SourceView, a source code editor widget with syntax highlighting), support for GNU/Linux, MacOS X and Windows platforms, ...

The library has reached version 0.9.7, and thus is still largely under development. Support for using the library can be found on the Gtk2Hs mailinglist. When this article is published, a fully working 0.9.7.1 build for Windows should be available. Instructions on how to install Gtk2Hs on Windows are available on the website (`http://gtk2hs.sourceforge.net/archives/2005/02/17/installing-on-windows`). The API is quite useful already, although it doesn't contain all the desired information.

When talking about a GUI library, it is often useful to first explain some of the used terminology, as we do not expect everybody to be familiar with several of the UI terms we use in this article, such as (cfr. Wikipedia):

- widget: a component of a graphical user inferface that the user interacts with. Examples: button, label, scroll bar, ...
- container: a widget which is able to contain other widgets
- box: a container which aligns all of its widgets either in a vertical or horizontal way

To use Gtk2Hs in a Haskell program, you should install Gtk2Hs and import the Gtk2Hs modules you need. For the Windows platform (and maybe others too), this installation will require Gtk+ to be installed (to install Gtk+ on the Windows platform, see *gladewin32.sourceforge.net*), and to be able to use the Glade functionality, you should also have Glade installed (more information, see the 'Using Glade' section).

Once everything is installed properly, it should suffice to add an import statement at the beginning of the program code, i.e.:

```
import Graphics.UI.Gtk
```

This lets the Haskell compiler know we want to use the functionality provided by the Gtk2Hs package.

Some people would argue that there are dozens of Haskell GUI libraries out there, so why would this one be any better than the rest? To point out the advantages of Gtk2Hs over other Haskell GUI libraries, we give a brief overview of what Gtk2Hs does better than most of the other libraries:

- **Glade support** First of all, using Glade you can design a GUI visually rather than having to write code. This allows one to follow the HIG (the Gtk/Gnome Human Interface Guidelines: `http://developer.gnome.org/projects/gup/hig`) much more easliy. It also allows Gtk2Hs to read the GUI definition at runtime. When the user wants to change some small things about the GUI, or even completely re-design it, no recompiling of the Haskell code is needed. Just make sure the new *.glade file has the same name as the last one, and contains the same widgets (with the same names) as the last GUI definition.
- **API reference documentation** An API is a tool which every serious developer needs. The Gtk2Hs tool which is available now, isn't complete yet, but several people are putting a lot of effort into it. The Gtk+ API (`http://gtk.org/api`) could also be of some help, since Gtk2Hs is a mapping of the Gtk+ functionality to Haskell.

- **Unicode support**
- **Memory managment**
- **Bindings for the Mozilla browser rendering engine**

To conclude this section, we list some of the other Haskell GUI libraries avaible.

- **wxHaskell** (*wxhaskell.sourceforge.net*) - built on top of wxWidgets, a comprehensive C++ library across all major GUI platforms, including Gtk, Windows, X11 and MacOS X
- **FranTk** (*haskell.org/FranTk*) - a declarative library for building GUIs in Haskell, running on top of Tcl-Tk (working via TclHaskell)
- **HToolKit** (*htoolkit.sourceforge.net*) - a portable Haskell library for writing graphical user inferfaces. To ensure portability, the library is built upon a low-level interface PORT, which is currently implemented for Gtk and Windows.
- **Other** (*haskell.org/libraries/#guigs*) - check here for information about other Haskell GUI library projects

The example program: Memory, the game

Before we show how to code the example we're using, let's see what it is supposed to do.

The Memory game is a simple card game. The goal of the game is find all pairs of equal images on the cards, which are all upside down when the game starts. We will allow the user to set the game level between 1 and 9. The level indicates how many pairs the deck of cards has: level 1 means just one pair (very easy), level 9 means nine pairs (quite 'hard'). When the user has set a level, he should be able to start a new game. We will use a control panel to provide this functionality.

In order to play the game, the player must be able to flip cards over, and see what picture is on them. Once he has flipped over two cards (and thus has tried to find a pair), the program should decide whether the cards match or not. When the player flips over another card, the game should visualise the match by means of some picture (in our case, a smiley icon), or just flip the cards back again when there was no match.

As a teaser, we show a screenshot of the Memory game which we have written. Some cards have been matched, two cards are flipped over (but no match is found), and the rest of the cards are upside down. On the right, the control panel is shown where the user can adjust the game level and/or start a new game.

The GUI: Using Glade

Glade is a tool to create XML descriptions of a graphical user interfaces. Using Glade, it is not necessary to write application code to construct GUIs. Moreover, it lets us redesign the GUI appearance without having to touch the code. No knowledge of Glade is needed to be able to understand this article. It was used to demonstrate one of many features of Gtk2Hs.

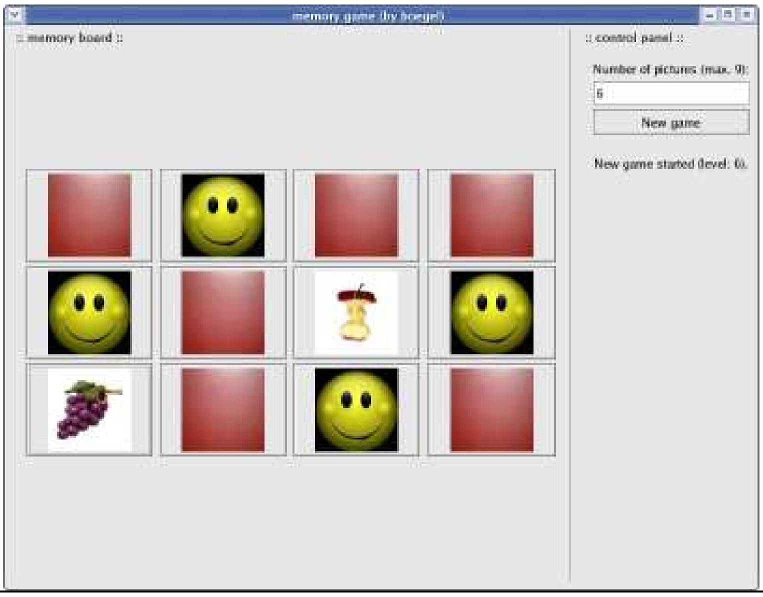

Figure 1: Screenshot of the Memory game

For more information about the use of Glade see `http://glade.gnome.org` for GNU/ Linux, or `http://gladewin32.sourceforge.net` for the Windows platform).

The code: reading the glade description

After using Glade to create the GUI, it is necessary to load the description of the various widgets in the GUI before they can be used, e.g. to define the communication between widgets, in an interactive application written in Haskell.

To make the Glade functionality available in Haskell, we should import the Glade module defined in the Gtk2Hs library first:

```
import Graphics.UI.Gtk.Glade
```

The first thing to do before loading the widgets defined in the Glade description, is to load the description itself. Because the *.glade file is needed to run the application, we include a check too see whether the description file can be found where it is expectedx. As such, we can inform the user with an appropriate error message if no conforming Glade file is available.

The load the file, we use the `xmlNew` function provided by the Glade module in the Gtk2Hs library. The `Maybe` module makes an excellent tool to help us check if the file was available. If it was, we use the description it contains. Otherwise, we throw an error, which will inform the user of what went wrong.

```
windowXmlM <- xmlNew "memory.glade"
let windowXml = case windowXmlM of
```

```
    (Just windowXml) -> windowXml
    Nothing -> error "Can't find the glade file \"memory.glade\"
                     in the current directory"
```

Now we are able to access the GUI description from within our code. This allows us the load the widgets defined in the description, using the `xmlGetWidget` function. When using Glade, one should try to define the widget as it should appear, only using Glade. That way, no adjustments have to be made when loading the widgets in Haskell.

```
window <- xmlGetWidget windowXml castToWindow "window"
   onDelete window deleteEvent
   onDestroy window destroyEvent
controlPanel <- xmlGetWidget windowXml castToVBox "control panel"
entry <- xmlGetWidget windowXml castToEntry "number of pictures"
button <- xmlGetWidget windowXml castToButton "new game button"
label <- xmlGetWidget windowXml castToLabel "message label"
   labelSetText label "\nNothing set."
boardAlignment <- xmlGetWidget windowXml castToAlignment "board content"
board <- xmlGetWidget windowXml castToVBox "cards"
```

As you can see, some extra code is needed besides simply loading the widgets.

Because a window doesn't react on click actions by default, we have to define what has to happen when the user closes the window. E.g. when a window is closed, the delete-event is thrown by Gtk+. For handling this event, the `onDelete` function is used to define which function should be executed when the window is closed.

We also have to define the `deleteEvent` function, which is called whenever a delete-event is thrown. Our implementation is quite simple: there is nothing we need to do when the user closes the window, besides killing the process showing it. To achieve this, we can simply return `False`, which will result in a destroy-event being thrown for the window. This behavior is the same as the default behavior, so if we wouldn't provide this, the window would be destroyed anyway. We mention it here explicitly to illustrate how the system works.

```
deleteEvent :: Event -> IO Bool
deleteEvent _ = do return False
```

In order to catch this destroy-event, we should use the `onDestroy` function (see above), in combination with the definition of the `destroyEvent` function (analogous to the delete-event). The latter just executes `mainQuit` (provided in the General module of Gtk2Hs), which will result in exiting the main loop (and thus killing the application).

```
destroyEvent :: IO()
destroyEvent = do mainQuit
```

The last line of extra code, is quite straightforward. It sets the default text on the messagelabel to “Nothing set.”, to show the user the application has just started, and no game level is set yet (which we will need in order to start a new game).

The code: what's after Glade

When all the widgets are loaded, there's still some work to do before we can show the GUI. Because we are building an interactive game application, we should be able to track what the user is doing (i.e. keep track of the game state). Haskell provides a handy 'tool' for that purpose: `IORef`. Because we are building a GUI, most of our code will be executed in the IO monad. Therefore, we can't just pass along some variable and expect every function to notice the change. The IORef module (available in the `Data` package of Haskell), provides a solution for this problem. We can read and write from/to a IORef, and thus share the variable with different functions.

The use of IORef is what distinguishes low/medium level libraries like Gtk2Hs and wxHaskell from high level libraries like FranTk and Fruit. The latter have various abstractions to avoid or hide the use of IORefs.

Throughout the program, we use a single IORef variable to keep track of the state. At the start of the program, we put this IORef in a well known state:

```
state <- newIORef (State Nothing Nothing 0)
```

In the IORef, we use a new data type `State`, that contains all the information we need about the game state, and which is declared as follows:

```
data State = State (Maybe ToggleButton)
                   (Maybe (Bool,ToggleButton, ToggleButton)) Int
```

The first of its arguments may contain a card which is flipped (or nothing when no card is flipped, that's why we use the Maybe monad). The second argument may contain a pair of cards which was tested for a match the last time (or again, nothing when no pair is available). The result of the match test is also available is this argument. The last argument contains the number of pairs of cards left to find, in order to be able to detect when the game is finished (i.e. when no pairs are left to find).

To complete the definition of `main`, we should add the following code:

```
onClicked button ...

widgetShowAll window
mainGUI
```

The `onClicked` part is discussed in the next section. The two last lines of code are needed to show all the widgets in the window, including the window itself (`widgetShowAll window`) and preparing the GUI for user interaction (`mainGUI`).

The code: setting up communication

The only piece of code of the `main` function we have not discussed yet, is the `onClicked` part mentioned earlier. This part defines what should happen when the "*New game*" button is pressed. Obviously, when there's more than one button in the GUI, the function `onClicked` should be defined for every button.

We first give the entire definition of the `onClicked` function, and then dissect is piecewise.

```
onClicked button $ do
    entryText <- entryGetText entry
    let text  = filter (not.isSpace) entryText
        check = (not $ null text) && null (tail text)  && isDigit
                                                           (head text)
    if (check) then do board <- vBoxNew True 1
                       cards <- startNewGame (read text ::  Int) label
                                                                  state
                       gameBoard <- createBoard cards state
                       boxPackStart board gameBoard PackNatural 0
                       children <- containerGetChildren boardAlignment
                       if (not $ null children) then containerRemove
                                                     boardAlignment
                                                     (head children)
                                                else return()
                       containerAdd boardAlignment board
                       widgetShowAll window
               else do children <- containerGetChildren boardAlignment
                       if (not $ null children) then containerRemove
                                                     boardAlignment
                                                     (head children)
                                                else return()
                       widgetShowAll window
```

Before the level entered by the user is used, it is paramount to check if the text entered in the entry field is correct.

We could avoid this with using a spin box (which would allow only values between 1 and 9) instead of an entry field. This would be much easier, but using an entry field allows us to show some additional aspects of Gtk2Hs (removing widgets from a box, what to do when unexpected user input is given, . . .).

First of all, we filter out all the spaces (to avoid unneccesary error messages).

```
    entryText <- entryGetText entry
    let text  = filter (not.isSpace) entryText
```

To check if the entered text is correct, we use the boolean variable `check`. First of all, we check if the text (without spaces) is 1 character long (because we only allow a level between 1 and 9). Then, we check if the character is a digit, using the `isDigit` function provided in the `Char` module.

```
check = (not $ null text) && null (tail text) && isDigit (head text)
```

Depending on the value of check, we decide what to do.

When the text entered is not correct (for example when a word is entered, or some non-numeric symbol), we should clear the game board (because it is possible another game was being played when the new game is started, and the current game should be removed from the board).

```
labelSetText label ("\nPlease enter a level between\n1 and 9 to set
                     the game level.")
children <- containerGetChildren boardAlignment
if (length children > 0) then containerRemove boardAlignment
                                   (head children)
                         else return()
widgetShowAll window
```

To clear the board, we simply remove all the widgets on it. In our impementation, the game board itself is created in a frame (a Gtk widget). On this frame, we have put a box container, in which all the cards are aligned. To clear the game board, it is thus sufficient to remove (and redraw) that box. For obtaining said box, the containerGetChildren function is used. This function returns a list of all the widgets that have been added to a container (an argument of the function). As shown above, the box is contained in a frame, and thus the box can be found among the children of the frame aligment. Since removing the box from the alignment actually equates to clearing the board, we must make sure the alignment is not empty before we try to obtain the first element of its children. Obviously, if that case, the code would try to take the head of an empty list, which would result in an abnormal termination of the program, or at least in a runtime error. Finally, to show the change, we add a call to the widgetShowAll function, with the entire window as its argument. The new widgets are created in a hidden state, so a function should be called in order to make them visible. We could have called widgetShowAll only on new widgets, but to avoid clutter, we chose the implementation shown above.

To inform the user what went wrong, we also show a suitable message on the provided label, using the labelSetText function.

When the entered text was correct, we need to show a new game board ready to play the game with the desired number of cards. In order to do this, we have to clear the board (analogous to the other case above) and add the new deck of cards.

```
let level = (read text :: Int)
labelSetText label ("\nNew game started (level: "++
                    (show level)++").")
cards <- buildNewGame level state
cardsBox <- fillBox cards state
children <- containerGetChildren boardAlignment
```

```
    if (length children > 0) then containerRemove boardAlignment
                                   (head children)
                             else return()
    containerAdd boardAlignment cardsBox
    widgetShowAll window
```

To ensure our code remains readable, we implemented the process of showing a new board in several separate functions. First we describe what happens, then we take a closer look at the steps taken to get the desired result.

The game level is entered as text, i.e. a string, and thus must be converted to an integer value. Next to that, we want to show a nice message on the label we provided for this purpose. The `buildNewGame` function yields the requested deck of cards. This deck is then used to build the box, which was mentioned earlier and which will contain the game cards. Building this box is done by the `fillBox` function. It is paramount that we pass along the IORef representing the game state, because this state will be needed by the function that deals with 'card clicks'.

Now, let's look at the definition of the `buildNewGame` and `fillBox` functions.

```
    buildNewGame :: Int -> IORef State -> IO Board
    buildNewGame n state = do
                    writeIORef state (State Nothing Nothing n)
                    let imagesPart = take n getImages
                        images     = imagesPart ++ imagesPart
                    case n of
                       1 -> return (Board 2 1 images)
                       2 -> return (Board 2 2 images)
                       ...
                       9 -> return (Board 5 4 images)
```

The `buildNewGame` function returns all the information needed to visualize the game board with a certain level (the number of different card pairs). For that purpose, we defined a `Board` datatype, containing the number of rows and columns of the game board, and a list of the names of the images for every card on the game board.

```
    data Board = Board Int Int [String]
```

First, `buildNewGame` sets the game-state to a well-known default value, and builds a list of image-names, as many as needed according to the chosen game level. To keep the example simple, we use a function which returns a static list of image-names.

```
    getImages :: [String]
    getImages = ["1.jpg","2.jpg","3.jpg","4.jpg","5.jpg","6.jpg",
                 "7.jpg","8.jpg"," 9.jpg"]
```

Depending on the chosen game level, the board layout will be chosen (the number of rows/columns).

The other function we need to define, is the `fillBox` function.

```
fillBox :: Board -> IORef State -> IO VBox

fillBox (Board w 1 list) state = do
                                    ...
fillBox (Board w 2 list) state = do
                                    ...

fillBox (Board w 3 list) state = do
          vBox <- vBoxNew True 1
          addHBoxToVBox w vBox (take w list) state
          addHBoxToVBox w vBox (take w (drop w lis t)) state
          addHBoxToVBox w vBox (drop (2*w) list) s tate
          return vBox

fillBox (Board w 4 list) state = do
                                    ...
```

This function will use the list of image names in the `Board` datatype to fill the box that represents the game board. The definition of `fillBox` depends on the number of rows needed to represent the game board. Every board consists of a vertical box containing a number of horizontal boxes of equal width. We show the definition of `fillBox` for a game board with 3 rows. A new vertical box is created, and then the horizontal boxes are added using the `addHBoxToVBox` function. We have to make sure we provide the right sublist containing the image-names.

```
addHBoxToVBox :: Int -> VBox -> [String] -> IORef State -> IO()
addHBoxToVBox w vBox list state = do hBox <- hBoxNew True w
                                     fillRow hBox list state
                                     boxPackStartDefaults vBox hBox
```

The `addHBoxToVBox` function just creates a new horizontal box of given width `w`, fills it with togglebuttons (using the `fillRow` function), and adds it to the vertical box also provided.

```
fillRow :: HBox -> [String] -> IORef State -> IO()
fillRow box [] state = do return ()
fillRow box (l:ls) state = do button <- toggleButtonNew
                              widgetSetName button $ l
                              containerSetBorderWidth button 2
                              image <- imageNewFromFile "back.jpg"
                              containerAdd button image
```

```
                                    onToggled button (buttonToggled button state)
                                    boxPackStartDefaults box button
                                    fillRow box ls state
```

The `fillRow` function creates a new togglebutton for every string in the given list with image-names. The name of the button is set to the image name (using `widgetSetName`), so when the togglebutton is clicked, we are able to show the image 'hidden' behind it. To start with, a default image is added to the togglebutton. We also define which function should be executed when the button is toggled (`buttonToggled`, see below), and of course the button is added to the horizontal box provided.

The code: playing the game

```
buttonToggled :: ToggleButton -> IORef State -> IO()
buttonToggled button stateRef = do
                  state <- readIORef stateRef
                  name <- widgetGetName button
                  pressed <- toggleButtonGetActive button
                  treatClick stateRef name button state pressed
```

The `buttonToggled` function just collects some information about the button which was toggled: the name of the button and the state of the button (pressed/unpressed). We also need the current game state. The actual actions which need to be executed when a button was toggled, are defined in the `handleClick` function.

```
handleClick :: IORef State -> String -> ToggleButton
                           -> State -> Bool -> IO()
```

Because this function has several cases, we will treat them one by one.

```
handleClick _ _ _ (State _ _ (-1)) _ = return ()

handleClick _ _ _ (State Nothing (Just (True,p1,p2)) 0) _ =
              do showImageOnButton p1 "found.jpg"
                 showImageOnButton p2 "found.jpg"

handleClick _ "found" _ _ _ = return ()
```

In `handleClick`, we distinguish several cases, handled separately by using pattern matching on the arguments of the function. The first case, where the total number of cards left in the game state is set to -1, is used when the game board should not react to any clicks. This is necessary when one button toggle results in 'un-toggling' another button, otherwise the un-toggling would trigger another execution of `handleClick`. When the the number of pairs to match is zero, and the last attempt to match succeeded, the game is finished. Here, we just make sure that the last pair of images are also replaced by smiley faces, but other actions can be added (showing a dialog box, adjusting the message in the control panel, . . .). The third case is executed when a button is toggled which contains a card that had already matched. Here, no changes must be made to the game state or the button which was clicked.

```
handleClick ref name button (State (Just lastButton) _ tot) False =
                        do writeIORef ref (State Nothing Nothing tot)
                           showImageOnButton button "back.jpg"
```

When a button is clicked, but the user decides to choose another card to start with, i.e. he clicks the same button again, the state should be set to the state the game was in before to the first click. Obviously, this case must be handled before the others, otherwise a match will be found, which is clearly wrong.

```
handleClick ref name button (State Nothing Nothing tot) _ =
                do writeIORef ref (State (Just button) Nothing tot)
                   showImageOnButton button name
```

This case occurs when a button is clicked while the game is in the 'empty' state. When this happens, the game is adjusted (it should show which button is currently clicked), and the image which belongs to the clicked button is shown, using the `showImageOnButton` function. This last function is discussed at the end of this section.

```
handleClick ref name button (State Nothing (Just (False,p1,p2)) tot) _ =
                do writeIORef ref (State (Just button) Nothing tot)
                   showImageOnButton button name
                   showImageOnButton p1 "back.jpg"
                   showImageOnButton p2 "back.jpg"

handleClick ref name button (State Nothing (Just (True,p1,p2)) tot) _ =
                do writeIORef ref (State (Just button) Nothing tot)
                   showImageOnButton button name
                   showImageOnButton p1 "found.jpg"
                   showImageOnButton p2 "found.jpg"
```

When the last attempt to match two images failed, and a new button was clicked, the first case will be executed. The game state is adjusted, so the currently clicked button is in it, and the last attempt to match is removed. The image of the clicked button is shown, and the images of the last two buttons which were clicked are reset to the default image.

When the last attempt did succeed, the same actions will be executed, but instead of resetting the images of the clicked button to the default one, the images are set to a 'found'-image (i.e. a smiley face).

```
handleClick ref name button (State (Just prevBut) _ tot) True =
      do last <- widgetGetName prevBut
         let matched = (name == last)
         writeIORef ref (State Nothing Nothing (-1))
         toggleButtonSetActive False button
         toggleButtonSetActive False prevBut
         prevName <- widgetGetName prevBut
         showImageOnButton button name
         if (matched) then do
               widgetSetName button "found"
               widgetSetName prevBut "found"
               writeIORef ref (State Nothing
                                     (Just (matched,button,prevBut))
                                     (tot-1))
               if (tot-1 == 0)
                  then toggleButtonSetActive True button
                  else return ()
         else writeIORef ref (State Nothing
                                    (Just (matched,button,prevBut))
                                     tot)
```

When some button has already been clicked, and a second is clicked, the definition of `handleClick` above is used. Because the toggle of both buttons is set to `False`, we have temporarily adjusted the game state, so no clicks will be accepted. The image of the second button is shown, so the user can see if the match succeeded or failed. Then, using the names of the buttons, we check if the match succeeded. When it did, the name of the button is adjusted to 'found', and the state is adjusted accordingly. When this matched pair was the last pair (which we can check using the number of pairs left to match), we force a button click, to make sure the 'found'-image is shown on both buttons. When the match failed, all we have to do is adjust the game state. Because the name of the buttons wasn't changed, the images will be reset to the default image when the next button is clicked.

To conclude the discussion of the code, we show the function, which is used to show a certain image on a button.

```
showImageOnButton :: Button -> String -> IO()
showImageOnButton button file = do
       children <- containerGetChildren button
       containerRemove button (head children)
       image <- imageNewFromFile file
       containerAdd button image
       widgetShowAll button
```

Because another image is already added to the button, we will remove it first, analogous to the way we removed the game board when a new game is started. Using the string which contains the path of the new image to be shown, we obtain the desired image, add it to the button, and show the new widgets added to the button using `widgetShowAll`.

The code: playing with efficiency

Because the example application we have written is quite small, efficiency isn't a real issue. Still, we would like to show how to improve the efficiency of our application.

The problem is that each time a button is pushed, the image the button hides is loaded from disk. Because our images are small, no real delay can be noticed. To avoid the images being loaded from disk every time, we could load them once, when the application is started. Then, when a button is pushed, we only have to replace the current image with another image already loaded.

Another thing we can improve, is the way how we change an image shown on a button. In the implementation above, we explicitly remove the current image, and add a new image to the button. A better way would be to change the image, rather than replacing it.

To kill two birds with one stone, we use the `Pixbuf` datatype, which contains all the information needed to create an image.

Loading all the images before they are needed, requires a way to make the loaded images accessible when needed. One way is to use a map, which is built at the beginning of the program:

```
type Images = [(String, Pixbuf)]
```

Now, when we need to 'load' an image, we can use the `lookup` function, which is defined in the Haskell Prelude.

Because we are using `Pixbuf`, there is no need to replace the image on a button, we can just change it. To illustrate, we show how the `showImageOnButton` function could look like:

```
showImageOnButton :: ToggleButton -> Images -> String -> IO ()
showImageOnButton button images imageName = do
  let Just image = lookup imageName images
  imageWidget <- liftM castToImage $ binGetChild button
  imageSetFromPixbuf imageWidget image
```

This implementation is both simpler and cheaper in terms of resources. Credits go to Duncan Coutts for this suggestion.

The game: really playing it

As you may have noticed when trying to play the game, it is quite easy to play, too easy really. The reason is simple: the list containing the image-names, and thus the list which determines the sequence of the cards on the board, is never shuffled. In order to implement the game as it is meant to be played, we should provide a `shuffle` function, which simply shuffles the list containing the image-names. We didn't bother to implement such a function, because it has nothing to do with the Gtk2Hs functionality.

Conclusion

Gtk2Hs is a powerful, user-friendly Haskell GUI library. The use of Glade to create a GUI allows the developer of the application to concentrate on the interaction part of the application, instead of making sure it looks good (just try to create the same layout as in the memory game, using only Haskell code, you'll see what I mean). The only drawback when using Glade, is that the *.glade file should always be available, which makes distribution of the application more difficult.

The current API available is very useful already, and since Gtk2Hs hasn't reached 1.0 yet, it will only improve. This article could serve as a Gtk2Hs tutorial for people who are not familiar with Gtk2Hs, and hopefully is a stimulation for other people working with Gtk2Hs to write a tutorial of their own. This way, the support for Gtk2Hs will increase, which will stimulate the growth of the library.

As a disclaimer, I would like to state the code isn't meant to be the most efficient code possible, neither to be bugfree. Improvements, suggestions and comments are always welcome, and the code is free to use in any way.

I hope this article has convinced the reader of the benefits of using Gtk2Hs as a Haskell GUI library, and has contributed to its popularity. Special thanks to Shae Matijs Erisson (the editor), Duncan Coutts (one of the Gtk2Hs people who made some suggestions) and Andy Georges (who proofread this article, and suggested a lot of improvements, mostly language related).