Caio Cesar Dionello **11080215**
Ricardo Rodrigues de Mendonça **11038614**

Tutorial de implementação da Biblioteca GTK2HS

Resumo
Este tutorial tem como objetivo introduzir e exemplificar o desenvolvimento de uma interface gráfica através da linguagem de programação funcional Haskell, com a utilização da biblioteca GTK2HS.

Introdução

O que é Haskell?

Uma linguagem de programação puramente funcional, desenvolvida em 1990, primeiramente com propósito somente acadêmico, mas se mostrando bastante efetiva na utilização em geral. Sendo funcional, é baseada no conceito de funções que somente recebem entrada e processam sua saída. Recomenda-se para a utilização deste tutorial um conhecimento médio em sua utilização, principalmente sobre conceitos que podem aparecer corriqueiramente: Avaliação preguiçosa, compreensão de listas, e Monads.

O que é GTK+?

Uma biblioteca multiplataforma destinada ao desenvolvimento de interface gráfica através da linguagem C++. É um software livre, com licença GNU. Apresenta interface de comunicação com diversas outras linguagens.

O que é GTK2HS?

Uma biblioteca de interface gráfica desenvolvida para Haskell, construída em cima da biblioteca GTK+. Suas funcionalidades utilizam ligações com as funções disponibilizadas pelo GTK+, com o mesmo poder de produções. É multiplataforma e disponibiliza tanto produções simples como diversas ferramentas para o desenvolvimento de GUI’s para aplicações.

Instalação

A instalação da biblioteca foi realizadas no Ubuntu 16.04 LTS, instalando-se o pacote libghc-gtk-dev através do comando

```
sudo apt-get install libghc-gtk-dev
```

Instruções para uma construção com mais recursos pode ser consultada através do link <https://wiki.haskell.org/Gtk2Hs/Installation>.

1. Considerações iniciais e estruturação

1.1 Construção básica

A interface gráfica desenvolvida em GTK é orientada a evento, ou seja, durante sua execução, o programa nada fará até a ocorrência de uma entrada de dados pelo movimento do mouse, clicar de uma tecla, ou qualquer outro acontecimento interno ao sistema; essas ocorrências são denominadas sinais. Então, tal ocorrência será processada, podendo ou não gerar uma saída ou procedimento, conforme o que fora definido pelo desenvolvedor.

Os elementos que compõe a interface são chamados *widgets*; desde um botão a uma janela e *container*, que veremos abaixo, são denominados *widgets* da interface.

Toda interface gráfica do usuário executa dentro da função ‘main’ do programa. Abaixo está descrito os recursos mínimos a serem utilizados para criação de uma simples interface gráfica, uma janela, com a explicação de cada instrução:

```
import Graphics.UI.Gtk        --importa todos os dados e funções da biblioteca

main :: IO ()                 --Declaração da função principal, ‘main’
main = do                     --Início do bloco ‘do’
      initGUI                 --Inicializa a GUI. Deve ser chamado antes de
                              --qualquer programa de interface gráfica,
                              --carregando todas as ferramentas do GTK.
      window <- windowNew     --Cria um widgetWindow, guardando seu valor
                              --em ‘window’
      widgetShowAll window    --Recursivamente mostra o widget ‘window’ e
                              --todos os seus filhos associados.
      MainGUI
```

O último procedimento inicializa o *loop* principal específico da GUI entre ‘initiGUI’ e ‘mainGUI’. Essa função faz com que o processo durma até que um sinal esperado seja disparado; assim que recebido o sinal, o processo então realiza o procedimento associado. Todas as instruções relativas à interface gráfica devem permanecer entre ‘initiGUI’ e ‘mainGUI’.

Outros elementos relevantes

O programa acima contém os recursos mais básicos necessários para criação de uma GUI. Entretanto, ao fechar a janela, o processo não finalizará a execução. O evento de acionamento do botão ‘fechar janela’ é tratado pela função abaixo:

```
--onDestroy :: WidgetClass w => w -> IO () -> IO (ConnectId w)

onDestroy window mainQuit
```

No GTK, os *widgets* possuem atributos que podem ser alterados ou consultados, de acordo com sua classe ou superclasse da qual deriva, utilizando as funções abaixo:

```
set widgetClass [firstAttribute:=value1, secondAttribute:=value2,
         thirdAttribute:=value3]
x <- get widgetClass first Attribute
```

Os atributos utilizados durante a construção dos exemplos estão disponíveis à consulta no fim deste tutorial, além de outros que foram considerados importante para serem citados, para facilidade do entendimento dos programas aqui construídos.

1.2 Estruturação

A organização especial dos *widgets* inseridos na tela podem ser realizadas de duas maneiras: utilizando o conceito de caixas e empacotamento, ou estruturando tabelas.

1.2.1 Caixas e empacotamento

A ideia principal deste conceito é criar estruturas chamadas caixas (‘boxes’) que podem conter *widgets* e outras caixas. Assim, pode-se realizar o empacotamento sucessivo de caixas, formando a uma estrutura gráfica abstrata, cujo formato será moldado pelas widgets que foram inseridas.

Dois tipos de caixas podem ser criadas: HBox, onde os *widgets* filhos serão inseridos no sentido horizontal, ou VBox, onde serão inseridos no sentido vertical. A inserção de outros elementos às HBox’s e VBox’s pode ocorrer respectivamente de duas formas: da esquerda para direita/ de cima para baixo, através da função boxPackStart, ou da esquerda para direita/ de baixo para cima, utilizando boxPackEnd. Abaixo consta a assinatura das funções para criação de Boxes, e inserção de elementos:

```
hBoxNew :: Bool         --homogeneidade dos elementos que a compõe
          -> Int        --distância em pixels padrão entre os elementos
          -> IO HBox    --retorno da função, uma nova HBox
```

analogamente,

```
vBoxNew :: Bool -> Int -> IO HBox
```

Para inserção de elementos:

```
boxPackStart :: (BoxClass self, WidgetClass child)
               => self              --Box que receberá o novo elemento
                -> child            --Elemento a ser inserido.
                -> Packing    --Tipo de empacotamento adotado. Os tipos serão
                              --comentados abaixo.
                -> Int        --Espaçamento em pixels do próximo elemento que
                              --será inseridos
                -> IO ()
```

analogamente,

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

'Packing' é o argumento da função acima que define como os elementos serão dispostos na caixa, especialmente quando a janela for redimensionada. Abaixo segue os valores possíveis e seus significados:

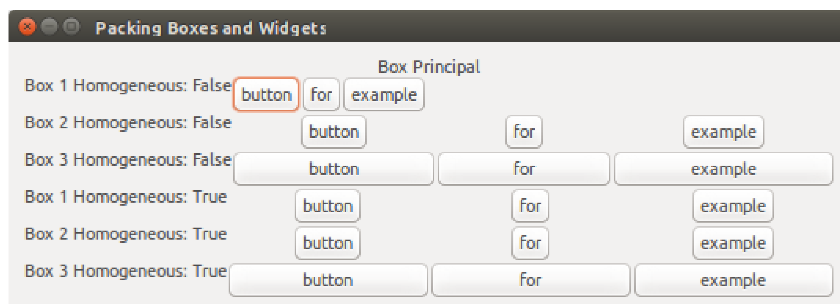
```
data Packing =
      PackRepel         --Elementos serão igualmente distribuídos pela caixa
      PackGrow          --Elementos ocuparão todo o espaço da caixa, crescendo
junto com a                        janela quando redimensionada
      PackNatural       --Elementos manterão sua distribuição na caixa conforme
pré-definido
```

Utilizaremos botões (Button) como *widgets* para exemplificar a disposição de objetos utilizando o conceito de empacotamento de caixas. As caixas serão nomeadas no nosso exemplo por uma etiqueta (label). Por enquanto, consideremos as funções

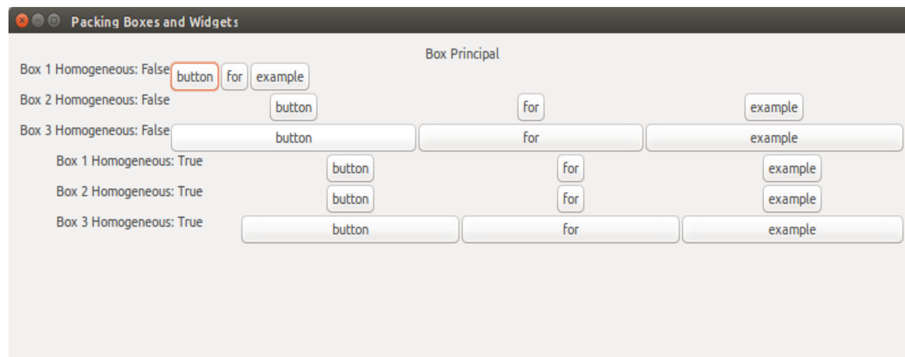
```
button <- buttonNewWithLabel “name1”
label  <- labelNew (Just "name")
```

para criação de botões e etiquetas, respectivamente. Mais informações sobre criação e manipulação destes elementos estão descritas após a próxima seção.

O exemplo cria 6 hBox contidas dentro de uma vBox, onde serão inseridos um label, indicando seu número e homogeneidade, e três botões por caixa, com os respectivos tipos de “Packing” Box 1 = PackNatural, Box 2 = PackRepel, Box 3 = PackGrow.



Janela criada



Janela após redimensionamento

```
import Graphics.UI.Gtk
import Control.Monad

main :: IO ()
main = do
  initGUI
  window             <- windowNew
  boxprincipal       <- vBoxNew False 0
  bplabel            <- labelNew (Just "Box Principal")
  set window [containerBorderWidth := 10,
              windowTitle := "Packing Boxes and Widgets",
              containerChild := boxprincipal]
  boxPackStart boxprincipal bplabel PackNatural 0
  boxesnh            <- createBoxes 3 hBoxNew False 0
  boxesh             <- createBoxes 3 hBoxNew True 0
  insertWidgets boxPackStart boxprincipal PackNatural 0 boxesnh
  insertWidgets boxPackStart boxprincipal PackNatural 0 boxesh
  buttons1            <- sequence [createButtons|x<-[0..6]]
  insertWidgets boxPackStart (boxesnh !! 0) PackNatural 0 (buttons1!!0)
  insertWidgets boxPackStart (boxesnh !! 1) PackRepel 0 (buttons1!!1)
  insertWidgets boxPackStart (boxesnh !! 2) PackGrow 0 (buttons1!!2)
  insertWidgets boxPackStart (boxesh !! 0) PackNatural 0 (buttons1!!3)
  insertWidgets boxPackStart (boxesh !! 1) PackRepel 0 (buttons1!!4)
  insertWidgets boxPackStart (boxesh !! 2) PackGrow 0 (buttons1!!5)
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI

createBox :: (BoxClass a) =>
             (Bool -> Int -> IO a) -> Bool -> Int -> [Char] -> IO a
createBox boxNew hom padd label = do box <- boxNew hom padd
                                     lab <- labelNew (Just label)
                                     boxPackStart box lab PackNatural 0
                                     return box
createBoxes :: (BoxClass a) =>
            Int -> (Bool -> Int -> IO a) -> Bool -> Int -> IO [a]
createBoxes n boxNew hom padd = do
            boxes <- sequence [boxNew hom padd |x<-[1..n]]
            labels <- sequence [labelNew (Just (
                                "Box " ++ (show x)++" Homogeneous: "
                                ++(show hom)))|x<-[1..n]]
            zipWithM (\bx lbs -> boxPackStart bx lbs PackNatural 0) boxes labels
            return boxes
createButtons ::IO [Button]
createButtons = do button1 <- buttonNewWithLabel "button"
                   button2 <- buttonNewWithLabel "for"
                   button3 <- buttonNewWithLabel "example"
                   return [button1, button2, button3]

insertWidgets :: (BoxClass a, WidgetClass b) =>
```

```
              (a -> b -> Packing -> Int -> IO())
              -> a -> Packing -> Int -> [b] -> IO[()]
insertWidgets boxPackf pBox pack padd buttons = do
                        mapM (\wdgt -> boxPackf pBox wdgt pack padd) buttons
```

Código da janela apresentada acima

1.2.2 Estruturação de tabelas

Os layouts dos *widgets* podem ser moldados também através do uso de tabelas, inserindo-os em suas células. As tabelas são criadas através da função abaixo:

```
tableNew ::      Int          --número de linhas
              -> Int          --número de colunas
              -> Bool         --Homogeneidade dos widgets inseridos
              -> IO Table
```

Para inserir itens na tabela, podemos utilizar a seguinte função:

```
tableAttach :: (TableClass self, WidgetClass child)
  => self         --tabela que receberá o objeto
  -> child              --objeto a ser inserido
  -> Int                --número da coluna do alinhamento esquerdo do objeto
  -> Int                --número da coluna do alinhamento direito do objeto
  -> Int                --número da linha do alinhamento superior do objeto
  -> Int                --número da linha alinhamento inferior do objeto
  -> [AttachOptions]    --opções para definir o comportamento no
                        --redimensionamento horizontal
  -> [AttachOptions]    --opções para definir o comportamento no
                        --redimensionamento vertical
  -> Int                --espaçamento horizontal da tabela
  -> Int                --espaçamento vertical da tabela
  -> IO ()
```

As opções de redimensionamento podem ser a seguintes:
Expand: tabela expande juntamente com a janela
Fill: *widget* ocupará todo o espaço disponível das células em que foi alocado
Shrink: *widget* irá diminuir junto com a janela

Alternativamente, pode-se utilizar a função

```
 tableAttachDefaults ::
 (TableClass self, WidgetClass widget) =>  self -> widget -> Int -> Int -> Int
-> Int -> IO ()
```

que utiliza os 6 primeiros argumentos da função anterior, inserindo *widgets* com espaçamento zero e opções padrão '[Expand, Fill]'

2. Botões

Botões são *widgets* que emitem sinal ao clique do mouse (ou alternativamente através de um mnemônico, pelo teclado), podendo apresentar diferentes funcionalidades. Aqui descreveremos os principais tipos de botões, suas funções e aplicações.

2.1 Buttons (botões simples)

Geralmente usados para realizar uma função quando pressionados. Para sua criação, podem ser utilizadas as funções:

```
buttonNew :: IO Button                    --Cria um botão
buttonNewWithLabel :: GlibString string
                      => string -> IO Buttton    --Cria um botão com etiqueta
```

```
buttonNewWithMnemonic :: GlibString string
                      => string -> IO Buttton --Cria botão com mnemônico
buttonNewFromStock :: StockId -> IO Button   --Cria botão com um ícone padrão do
GTK
```

Alternativamente, pode ser inserido uma caixa a um botão criado pela primeira das funções acima, através da função

```
containerAdd :: (ContainerClass self, WidgetClass widget)
               => self        --botão que receberá a caixa
                -> widget           --caixa a ser inserida
                -> IO ()
```

desta forma, pode-se criar um botão com vários elementos, como imagens e etiquetas.

O mnemônico descrito na terceira função aparece como uma etiqueta no botão, mas poderá ser acessado do teclado ao acionar a tecla ‘alt’ e digitar o caractere correspondente que será sublinhado no botão. É uma opção interessante para aumentar a acessibilidade da interface.

Os sinais emitidos através do acionamento dos botões podem ser tratados pelas seguintes funções:

```
onPressed :: ButtonClass self
      => self -> IO () -> IO () --aplica uma função quando botão é pressionado
onReleased :: ButtonClass self
      => self -> IO () -> IO () --aplica uma função quando botão é solto
onClicked :: ButtonClass self
      => self -> IO () -> IO () --aplica uma função quando botão é clicado
                                 --(pressionado e solto)
```

2.2 Botão alternável (toggle button)

Derivado do *widget* acima, entretanto mantém seu estado quando pressionado. Apresente três funções construtoras:

```
toggleButtonNew :: IO ToggleButton
toggleButtonNewWithLabel :: GlibString string => string-> IO ToggleButton
toggleButtonNewWithMnemonic  ::  GlibString  string  =>  string  ->  IO  ToggleButton
```

quanto ao seu acionamento ou estado, temos as seguintes funções:

```
onToggled :: ToggleButtonClass self
          => self -> IO () -> IO (ConnectId self)  --aplica uma função quando
                                                   --botão muda de estado
toggleButtonSetMode :: ToggleButtonClass self
          => self -> Bool -> IO ()  --Seleciona o estado do botão; True = ativo.

toggleButtonGetMode:: ToggleButtonClass self
                     => self -> IO Bool        --recupera o estado do botão.
```

2.3 Check buttons

Possuem a mesma funcionalidade que o *toggle button*, entretanto sua aparência visual é diferente. As funções de acionamento e recuperação/seleção de estados são as mesmas utilizadas pelo *widget* anterior; as funções de criaçã são as seguintes:

```
checkButtonNew :: IO CheckButton
checkButtonNewWithLabel :: GlibString string => string -> IO CheckButton
checkButtonNewWithMnemonic :: GlibString string => string -> IO CheckButton
```

2.4 Radio buttons

Similar aos *check buttons*, entretanto, somente um pode ser acionado no momento. Devido a essa propriedade, durante a criação de um novo botão é necessário associar a qual grupo ele pertence. A

criação do primeiro botão do grupo é similar aos dois *widgets* acima, através das 3 seguintes funções:

```
radioButtonNew :: IO RadioButton
radioButtonNewWithLabel :: GlibString string => string -> IO RadioButton
radioButtonNewWithMnemonic :: GlibString string => string -> IO RadioButton
```

A inserção de um novo botão que será associado a um grupo já existente pode ser feita através das seguintes funções:

```
radioButtonNewFromWidget :: RadioButton -> IO RadioButton
radioButtonNewWithLabelFromWidget :: GlibString string =>
                                       RadioButton -> string -> IO RadioButton
RadioButtonNewWithMnemonicFromWidget :: GlibString string =>
                                                  RadioButton  ->  string  ->  IO
RadioButton
```

Alternativamente, pode-se utilizar a seguinte função para associar um radioButton já criado a um grupo já existente, através da seguinte função:

```
radioButtonSetGroup
:: RadioButton          --Botão a ser associado
-> RadioButton          --Qualquer  membro do grupo que o botão será associado
-> IO ()
```

Outra função que pode apresentar suas utilidades, é recuperar um grupo de botões ao qual um botão está associado, o que pode ser realizado através da função

```
radioButtonGetGroup :: RadioButton -> IO [RadioButton]
```

2.5 Scale Buttons (widgets de escala)

Há dois tipos principais de *widgets* que disponibilizam uma faixa de valores para selecionar.: Os scaleButtons, derivado da classe botão, e os *widgets* HScale e VScale, que serão apresentados na seção seguinte.

Os scaleButtons são botões que mantendo a régua oculta ao usuário, apresentando-a somente após serem clicados; os principais métodos são:

```
scaleButtonNew :: GlibString string            --cria novo scaleButton
=> IconSize                              --tamanho do ícone
-> Double                          --valor mínimo
-> Double                          --valor máximo
-> Double                          --passo de mudança do valor ao ser alterado
-> [string]                              --ícones
-> IO ScaleButton

--trata o sinal de quando o valor é aumentado
scaleButtonPopdown :: ScaleButtonClass self => Signal self (IO ())
--trata o sinal de quando o valor é diminuido
scaleButtonPopup :: ScaleButtonClass self => Signal self (IO ())
--trata o sinal de quando o valor é mudado
scaleButtonValueChanged :: ScaleButtonClass self => Signal self (Double -> IO
())
```

Esse botão pode ser usado para implementar, por exemplo, o controle de volume da aplicação. Na verdade, tem uma classe derivada dos ScaleButton que já apresenta esse padrão; através da função

```
volumeButtonNew :: IO VolumeButton
```

é possível criar essa funcionalidade. Os métodos para tratar sinais são os mesmo descritos acima, da classe a qual deriva.

3. Widgets de ajustamento

Abaixo será apresentado os HScale e VScale, bem como o SpinButton. Estes dois primeiros são semelhantes aos apresentados na seção anterior (2.5 Scale Buttons), entretanto, apresentam conceitos de criação e diferentes, juntamente com o SpinButton. Eles fazem parte de uma classe de *widgets* com a finalidade de apresentar uma interface com opções de de ajustamento ao usuário.

A construção desses objetos podem ser derivados de um conjunto de ajustamento genérico pré-definidos, através do método adjustmentNew, definido abaixo. A utilização desse método é vantajoso na hora de desenvolver uma aplicação onde as característica desses ajustamentos podem aparecer de forma repetida, ou simplesmente para facilitar na manutenção, evitando-se alterar diretamente os *widgets* que a implementam. A finalidade de cada um dos argumentos podem depender do *widget* em que ele é aplicado.

```
adjustmentNew :: Double              --valor inicial da escala
              -> Double              --valor mínimo
              -> Double              --valor máximo
              -> Double              --incremento mínimo
              -> Double              --incremento máximo
              -> Double              --tamanho da área visível
              -> IO Adjustment
```

3.1 HScale e VScale

Os HScale e VScale, por sua vez, apresentam a régua de escala diretamente na janela, sem um botão intermediário, respectivamente no sentido horizontal e vertical. Podem ser criados através das funções

```
vScaleNewWithRange :: Double        --valor mínimo
                   -> Double        --valor máximo
                   -> Double        --passo de mudança do valor ao ser alterado
                   -> IO VScale

hScaleNewWithRange :: Double -> Double -> Double -> IO Hscale
```

Alternativamente, pode-se criar esses *widgets* através de um ajustamento pré-definido, simplesmente utilizando as funções abaixo.

```
vScaleNew :: Adjustment -> IO VScale

hScaleNew :: Adjustment -> IO Hscale
```

Dois métodos importantes para utilizar estes *widgets* são:

```
--faz uma ação quando é mudado o valor da escala
onValueChanged :: Adjustment -> IO () -> IO (ConnectId Adjustment)
--recupera o valor atual
adjustmentGetValue :: Adjustment -> IO Double
```

3.2 Spin Buttons

São pequenos campos de entrada, permitindo que usuário incremente ou decremente o valor dispostos através de duas setas, bem como permitindo que tal valor seja digitado diretamente. Podem ser criados através dos métodos abaixo:

```
spinButtonNew :: Adjustment   --tipo de ajustamento que o spin button usará
                -> Double     --especifica quanto o  valor irá mudar quando a
                                seta é clicada
                -> Int        --número de casa decimais que será apresentado
                -> IO SpinButton
```

Pode ser criado sem precisar desenvolver o ajustamento manualmente, através do método abaixo.

```
SpinButtonNewWithRange :: Double                --valor mínimo
                       -> Double                --valor máximo
                       -> Double                --passo do incremento
                       -> IO SpinButton
```

4. Labels

Um label é um simples texto escrito graficamente na interface. Tem um comportamento parecido com um *container;* pode ser inserido dentro de um botão ou de outro *container. Um l*abel pode ser criado pelas funções

```
labelNew :: GlibString string => Maybe string -> IO Label
labelNewWithMnemonic :: GlibString string => string -> IO Label
```

podendo sua justificação ser alterada, através da função

```
labelSetJustify :: LabelClass self => self -> Justification -> IO ()
```

Sendo Justification um tipo de data, com os seguintes valores possíveis:

```
data Justification =
              JustifyLeft      --esquerda
            | JustifyRight     --direita
            | JustifyCenter    --centro
            | JustifyFill      --justificado
```

5. Frame

Frame é um *widget* tipo container, que é destacado visualmente, sendo utilizado como um ornamento na interface. Pode ser criado pela função

```
frameNew :: IO Frame
```

onde os principais métodos aplicáveis são:

```
--insere uma etiqueta destacada na parte de cima do frame
frameSetLabel :: (FrameClass self, GlibString string) => self -> string -> IO ()

--recupera a etiqueta
frameGetLabel :: (FrameClass self, GlibString string) => self -> IO string

--define o ajustamento do conteúdo interno
frameSetLabelAlign :: FrameClass self
      => self
      -> Float    --alinhamento horizontal: 0.0 à esquerda, 1.0 à direita
      -> Float    --alinhamento vertical: 0.0 alinha abaixo, 1.0 acima
      -> IO ()
```

5. Text Entry (entrada de texto)

Este *widget* disponibiliza uma caixa de texto que pode ser inserido pelo usuário, com qualquer tipo de dado. Alternativamente, pode-se utilizá-la para criar um campo não editável, bem como uma entrada de senha, alterando-se os atributos do *widget* criado. Uma caixa de entrada de texto pode ser construída e manipulada através das funções abaixo.

```
--cria uma caixa para entrada de texto
entryNew :: IO Entry

--escreve na caixa
```

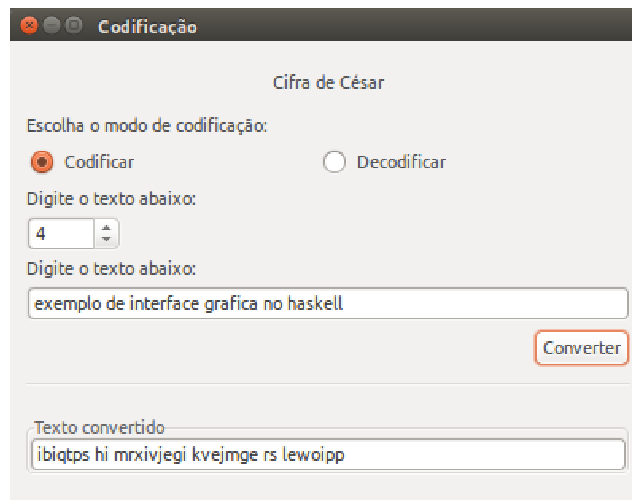
```
entrySetText :: EntryClass self => self -> String -> IO ()
--recupera o texto
entryGetText :: EntryClass self => self -> IO String
```

6. Exemplo de aplicação

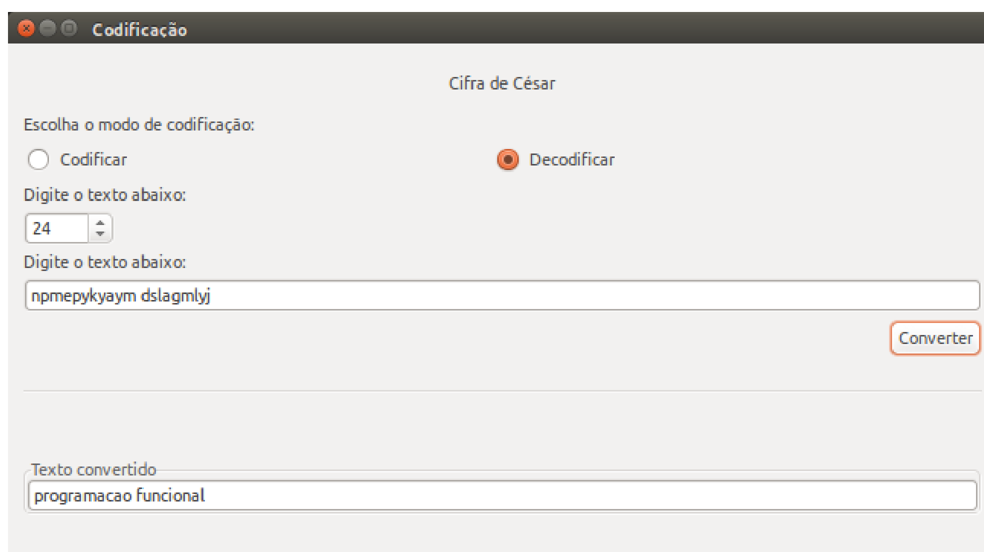
Utilizando os conceitos apresentados nesse tutorial, podemos desenvolver uma aplicação com funcionalidades, para que visualmente se veja o efeito do desenvolvimento de aplicações utilizando-se essa biblioteca.

Nesse exemplo, foi desenvolvido uma aplicação para Codificação da Cifra de César. O usuário pode digitar o texto a ser analisado no primeiro campo de entrada, escolhendo através dos Radio Buttons a opção se o texto será criptografado ou descriptografados. Assim, ao apertar o botão "Converter", o texto resultado será disponibilizado em outro campo de entrada, mas este não editável, que está inserido dentro de um Frame, com o título "Texto convertido". A chave da Cifra pode ser definida dentro de um Spin Button.

A disposição dos elementos foi desenvolvida utilizando-se o conceito de empacotamento de caixas, com diferentes tipos de empacotamento, que pode ser consultado no código fonte.



Aplicação – Cifra de César - Codificar



Aplicação – Cifra de César – Decodificar – Redimensionado

Abaixo está disponibilizado o código-fonte da aplicação:

```
import Graphics.UI.Gtk
import Data.Char

main :: IO ()
main = do
  initGUI
  window      <- windowNew
  pBox        <- vBoxNew False 5
  set window [windowTitle          :="Codificação",
              containerBorderWidth := 10,
              windowDefaultWidth   := 400,
              containerChild       :=pBox]
  title        <- labelNew (Just "Cifra de César")
  boxPackStart pBox title PackNatural 10

  box0         <- hBoxNew False 0
  boxPackStart pBox box0 PackNatural 0
  label1       <- labelNew (Just "Escolha o modo de codificação:")
  boxPackStart box0 label1 PackNatural 0

  box1         <- hBoxNew False 0
  boxPackStart pBox box1 PackNatural 0
  option1      <- radioButtonNewWithLabel "Codificar"
  option2      <- radioButtonNewWithLabelFromWidget option1 "Decodificar"
  boxPackStart box1 option1 PackNatural 0
  boxPackStart box1 option2 PackRepel 0

  box2         <- hBoxNew False 0
  boxPackStart pBox box2 PackNatural 0
  label2       <- labelNew (Just "Digite o texto abaixo:")
  boxPackStart box2 label2 PackNatural 0

  box3         <- hBoxNew False 0
  boxPackStart pBox box3 PackNatural 0
  spindAdj     <- adjustmentNew 0 (-1000) 1000 1 1 0
  key          <- spinButtonNew spindAdj 1 0
  boxPackStart box3 key PackNatural 0

  box4         <- hBoxNew False 0
  boxPackStart pBox box4 PackNatural 0
  label3       <- labelNew (Just "Digite o texto abaixo:")
  boxPackStart box4 label3 PackNatural 0

  box4         <- hBoxNew False 0
  boxPackStart pBox box4 PackNatural 0
  entry        <- entryNew
  boxPackStart box4 entry PackGrow 0

  box5         <- hBoxNew False 0
  boxPackStart pBox box5 PackNatural 0
  start        <- buttonNewWithLabel "Converter"
  boxPackEnd   box5 start PackNatural 0

  sep          <- hSeparatorNew
  boxPackStart pBox sep PackRepel 5

  box6         <- hBoxNew False 0
  boxPackStart pBox box6 PackRepel 10
  answerFrame  <- frameNew
```

```
  set answerFrame [frameLabel := "Texto convertido"]
  boxPackStart  box6 answerFrame PackGrow 0
  answer       <- entryNew
  set answer [entryEditable   := False]
  containerAdd answerFrame answer

  onClicked start (convert entry key option1 answer)

  onDestroy window mainQuit
  widgetShowAll window
  mainGUI

setOption :: RadioButton -> IO [Char]
setOption button = do
      opt <- get button buttonLabel
      return opt

convert :: Entry -> SpinButton -> RadioButton -> Entry -> IO ()
convert ent spinB opt ans = do
          textI      <- entryGetText ent
          key        <- get spinB spinButtonValue
          codeState  <- toggleButtonGetActive opt
          let mode   =  if codeState == True
                           then 1
                           else (-1)
          let textF  = encode (mode*(floor key)) textI
          entrySetText ans textF
          return ()

--caeser cipher functions
let2int :: Char -> Int
let2int c = ord c - ord 'a'

int2let :: Int -> Char
int2let n = chr (ord 'a' + n)

shift :: Int -> Char -> Char
shift _ ' '=  ' '
shift n c   = int2let newWord
  where
    newWord = newWordNum `mod` 26
    newWordNum = (let2int c) + n

encode :: Int -> String -> String
encode n xs = map (shift n) xs
```

Atributos dos *widgets.*

Caso os atributos tenham valores pré-definidos, estão descrito entre parênteses. O valor padrão do atributo, i.e. o valor assumido pelo atributo caso não tenha sido modificado estará sublinhado.

→ *WindowClass:*
  → *windowTitle: Título da Janela*
  → *windowType: Tipo da janela (WindowToplevel | WindowPopup)*
  → *windowResizable: Permissão de redimensionamento da tela (True | False)*
  → *windowDefaultWidth: Define largura padrão (v >= -1)*
  → *windowDefaultHeight: Define altura padrão (v >= -1)*
  (derivados da classe ContainerClass)
  → *containerBorderWidth: largura da borda em volta do Container filho (v >= 0)*
  → *containerChild: especifica qual widget será o Container filho.*
→ *Table*
  → *tableNRows: quantidade de linhas da tabela (v >= 0)*
  → *tableNColumns: quantidade de colunas da tabela (v >= 0)*
  → *tableRowSpacing: espaçamento padrão da linha (v >= 0)*
  → *tableColumnSpacing: espaçamento padrão da coluna (v >= 0)*
  → *tableHomogeneous: homogeneidade dos itens das células (True | False)*
→ *Button*
  → *buttonLabel: Etiqueta do botão*
  → *buttonLabel: Etiqueta do botão*
  → *buttonFocusOnClick: Se o botão segura o foco quando clicado (True | False)*
  → *buttonImage: Imagem a ser inserida no botão*
→ *Toggle Button*
  → *toggleButtonActive: Indica se o botão deve estar ativado (True|False)*
→ *Radio Button*
  → *radioButtonGroup: Grupo de botões ao qual o que recebe o atributo pertence*
→ *Scale Button*
  → *scaleButtonValue: Valor da escala (v >= 0)*
  → *scaleButtonSize: Tamanho do ícone (padrão: IconSizeSmallToolbar)*
  → *scaleButtonAdjustment: ajustamento da escala*
  → *scaleButtonIcons: ícones do widget*
→ *Frame*
  → *frameLabel: especifica o texto da etiqueta do frame*
  → *frameLabelXAlign: alinhamento horizontal (1>=v>=0, 0.5)*
  → *frameLabelYAlign: alinhamento vertical (1>=v>=0, 0.5)*
→ *Label*
  → *labelLabel: texto do label*
  → *labelJustify: justicação do texto (JustifyLeft | JustifyRight | JustifyCenter| JustifyFill)*
  → *labelAngle: ângulo de inclinação do label (360>=v>=0)*
→ *Text Entry*
  → *entryEditable: se o conteúdo é editável (True | False)*

→ *entryMaxLength: maior comprimento de texto (65535>=v>=0)*
→ *entryVisibility: insere caractere escondido caso falto – modo senha (True | False)*
→ *entryInvisibleChar: caractere que deve ser inserido no modo senha (*)*
→ *entryText: o conteúdo do widget*