Interfaces Gráficas em Haskell com WxHaskell

Tutorial

Elton M. Cardoso
Lucília Figueiredo

Índice analítico

Capítulo 1 1
Introdução 1
1.1 – WxWindows e FFI 1
1.2 – Criando uma janela 2
1.3 – Compilando e executando a aplicação: 3
1.4 – Atributos e Propriedades: 5
1.4.1 – Eventos e tratadores de eventos 7
1.4.1.1 – Eventos de mouse 7
1.4.1.2 – Eventos de teclado 10
1.4.1.3 – Outros eventos: 11
1.5 – Layouts 12
..... 15
1.6 – Variáveis Mutáveis: 17
2 – Controles de interface gráfica. 17
2.1 – Window 18
2.2 - Frames 19
2.3 – MDI Frames 22
2.4 - Panel 23
2.5 – Notebook 24
2.6 – Button 25
2.7 – TextCtrl 25
2.9 – Check Box 27
2.10 – Choice 28
2.11 – ComboBox 29
2.12 – ListBox 30
2.13 – RadioBox 31
2.14 – Spin Control 32
2.15 – Slider 32
2.16 – Gauge 33
2.17 – Static text 34
2.18 – Splitter Window 35
2.19 – Image List 36
2.20 – List Control 37
2.21 – Tree Control 39
2.22 – Timer: 41
2.23 – Menu 41
2.24 – Tool bar 42
2.25 – Status Bar 43
3 – Diálogos 43
3.1 – Diálogos predefinidos: 44
3.1.1 - Mensagens 44
3.1.2 – Arquivos: 44
3.1.3 – Miscelânia: 45

3.2 – Escrevendo os próprios diálogos: 46
4 – O módulo Draw 47
4.1 – Atributos de DC oriundos da classe Draw 48
4.2 – Atributos de DC oriundos da classe Brushed 49
4.3 – funções para desenho de formas básicas: 50
5 – Processamento de Som e imagens 51
5.1 – Recursos para imagens: 51
5.2 – Recursos para Som: 52

Capítulo 1

Introdução

1.1 – WxWindows e FFI

A biblioteca *WxHaskell* [2] provê uma interface para o uso, em programas Haskell [1], de classes da biblioteca *WxWindows* [3], escrita na linguagem C++. As classes da biblioteca *WxWindows* provêm funcionalidades para criação de componentes de interfaces gráficas (tais como *widgets* (janelas), containers (componentes usados para conter outros componentes gráficos), buttons (botões) etc), assim como para tratamento de eventos que ocorrem nessas interfaces gráficas.

Para prover essa interface, a biblioteca *WxHaskell* é construída com base em funções disponíveis na biblioteca FFI (*Foregin Function Interface*) [4], a qual possibilita o uso de código escrito em outras linguagens, dentro de programas Haskell. A biblioteca FFI provê uma interface programável entre o contexto de Haskell e o contexto de uma linguagem externa (tal como C++, Java, Pascal etc). Como resultado, *threads* de Haskell podem fazer acesso a dados de um contexto externo, assim como invocar funções que serão executadas nesse contexto externo, e vice-versa. Neste tutorial, não iremos entrar em detalhes sobre a biblioteca FFI e suas aplicações. Entretanto, pode ser útil ter uma compreensão mínima sobre como *WxWindows* foi "portada" para Haskell, o que comentamos a seguir.

De maneira bastante simplificada, podemos dizer que *WxHaskell* faz um mapeamento dos tipos da biblioteca *WxWindows* para tipos de dados em Haskell. Note que, como C++ é uma linguagem orientada a objetos, a biblioteca *WxWindows* pode possuir variáveis cujo conteúdo tem tipo definido dinamicamente, em face do mecanismo de subtipos e ligação dinâmica existente em linguagens orientadas a objetos. Em contraposição, Haskell possui um sistema de tipos totalmente estático. Como solução, a implementação de *WxHaskell* introduz o tipo abstrato `Object a`, onde o parâmento "a" pode ser usado para expressar relações da hierarquia de classes de *WxWindows*. Além desse mapeamento entre tipos, *WxHaskell* cria tipos abstratos correspondentes às classes C++ da biblioteca *WxWindows* (tais como *Button*, *Window*, *EvtHandler* etc.) e mapeia as relações de hierarquia existentes entre estas classes.

Note, portanto, que, em um programa Haskell que utiliza a biblioteca *WxHaskell*, a criação e manutenção de uma interface gráfica não é executada no contexto de Haskell, mas sim no contexto externo, de C++. Por outro lado, a interação do programa com essa interface é feita por meio dos tipos e construções naturais de Haskell.

1.2 – Criando uma janela

Como um ponto de partida para explorar a biblioteca *WxHaskell*, apresentamos a seguir um exemplo que ilustra a criação de uma janela com um único botão, que permitirá fechá- la.

```
module MyFirstProgram where

{-
  importa a biblioteca gráfica WxHaskell.
-}
import Graphics.UI.WX

{-
  função principal que inicia a interface gráfica
-}
main :: IO ()
main = start interface

{-
  função que gera a interface gráfica propriamente dita.
-}
interface :: IO ()
interface
 = do f ← frame [text := "Criando uma janela simples"]
      b ← button f [text := "Fechar", on command := close f]
      set f [layout := margin 4 (floatCentre (widget b) ),
             clientSize := sz 280 100]
```

A função main chama a função start :: IO a IO (), que "inicia" a interface gráfica.

A função interface consiste no *seqüenciamento* de ações que criam os objetos de interface, com propriedades pré- definidas. O primeiro objeto criado é um *frame* (de tipo Frame), um tipo de *container* para outros objetos. A assinatura função que cria o *frame* é:

```
frame ::  [Prop (Frame () )]   IO (Frame () )
```

Esta função tem como parâmetro uma lista de propriedades de Frame – propriedades e atributos de um Frame são discutidas mais adiante – e retorna

um valor do tipo IO (Frame ()). Este tipo indica que o valor retornado, quando avaliado, resulta na ação de exibir na tela do computador o Frame encapsulado no valor deste tipo.

No corpo da função interface, o operador (←) é então usado desencapsular o Frame e atribuir o valor correspondente à variável f.

Em seguida. É criado um botão, por meio da função button:

```
button :: Window a    [Prop (Button () ) ]    IO (Button ())
```

Por fim, especificamos o *layout* da janela principal. Este é um passo obrigatório pois, caso não seja realizado, os *widgets* serão "empilhados" no canto superior esquerdo da janela.
O *layout* é um atributo comum a todos os *containeres* de *widgets*. No nosso exemplo, utilizamos a função widget, que extrai um *layout* de um controle:

```
widget :: Widget w => w    Layout
```

A assinatura da função widget indica que, para extrair um layout de um objeto, este deve ser uma instância da classe Widget. Na prática, todos os objetos criados a partir da classe Window são instâncias de Widget. Note que, aqui, estamos falando de classes de Haskell, e não de C++. Lembre-se que, em Haskell, classes são usadas para especificar o tipo de funções *sobrecarregadas*, e instâncias dessas classes constêm definições dessas funções para tipos específicos. Em contraposição, em linguagens orientadas a objetos, classes são usadas para definir tipos de objetos em um contexto de *polimorfismo de subtipagem*, e não de polimorfismo de sobrecarga.

Uma vez extraído o layout, é usada a função

```
floatCentre :: Layout    Layout
```

para centralizar o layout na janela. Ao layout resultante, adicionamos uma margem de 4 pixels, pela aplicação da função margin, com argumento igual a 4.

1.3 – Compilando e executando a aplicação:

Depois de digitar e salvar o programa, vamos executá-lo. Recomendamos o uso do compilador Haskell GHC-6.2.2 [1]. Se a biblioteca *WxHaskell* estiver corretamente instalada em seu sistema, basta abrir um prompt de comando no diretório onde você salvou seu arquivo e digitar:

ghci -package wx *NomeDoPrograma*.hs.

É obrigatório informar o parâmetro -package wx, para que o GHC saiba que deve ser usada a biblioteca *WxHaskell*. Se você estiver ussando algum ambiente de programação (p.ex., JCreator), poderá configurá-lo para iniciar o GHC já com este parâmetro. A janela do GHC terá então a seguinte aparência:

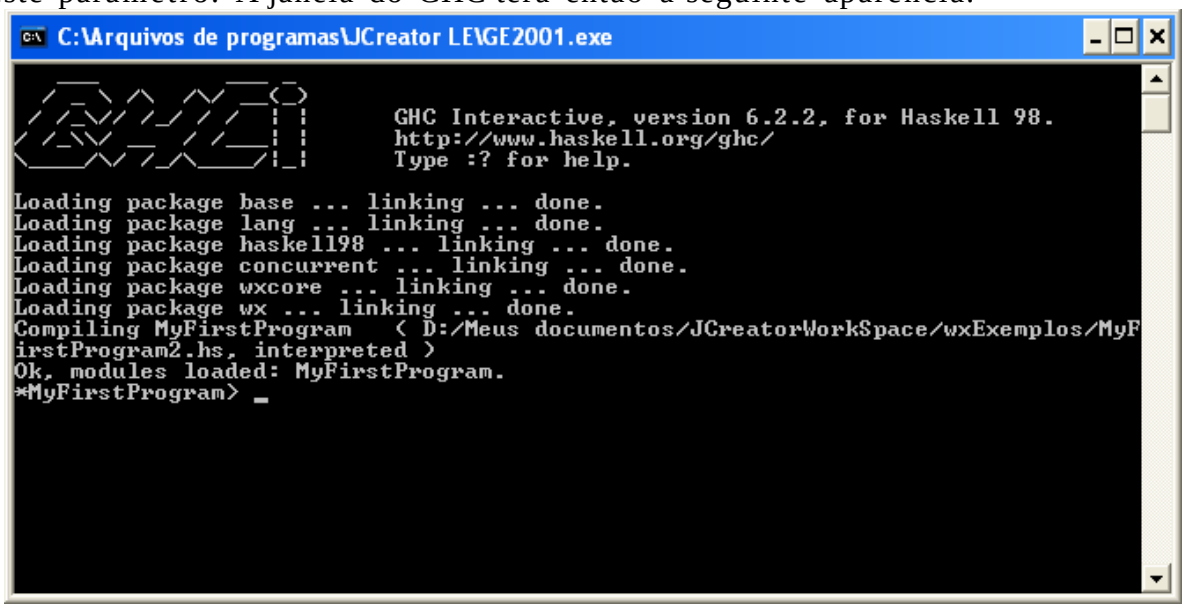

Figura 1 - Janela do GHCi como o programa carregado.

Note que os módulos da biblioteca *WxHaskell* foram carregados. Basta digitar main e teremos o resultado:

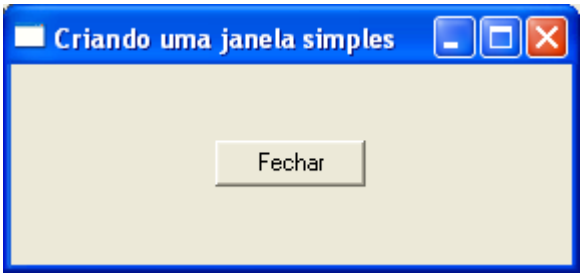

Figura 2 - Resultado da execução do programa.

Também é possível gerar código binário a partir do programa, mas isto irá depender do tipo e da versão do compilador que você estiver usando. No caso do GHC-6.2.2, deve ser usado o seguinte comando, para compilar o programa:

ghc -package wx -main-is MyFirstPogram --make MyFirstPogram.hs

O compilador irá então gerar os códigos objetos e ligá-los em um código executável. Em geral, a biblioteca é de *linkagem* estática, o que significa que o

programa executável conterá todas as funções da biblioteca, mesmo as que não são usadas no programa. Isto faz com que o código executável fique enorme! Uma solução para esse problema é usar o *strip* [referência], um programa capaz de vasculhar o código executável e remover símbolos que não são utilizados .

1.4 – Atributos e Propriedades:

Objetos em geral possuem atributos, que são representados por variáveis (de instância) de um determinado tipo, podendo assumir certos valores. Em *WxHaskell* atributos estão definidos da seguinte forma:

```
data Attr w a
```

onde w é um Widget (um objeto de interface) e *a* é o tipo do atributo em questão. Esta construção pode, portanto, ser interpretada da seguinte maneira: o Widget w possui um atributo do tipo a. Um atributo muito comum é o text, com o seguinte tipo:

```
text :: Attr  (Window a) String
```

Isto significa que todo objeto derivado da classe Window possui um atributo text. Os atributos podem ser somente de leitura, somente de escrita ou de leitura-escrita, o que é mais comum.

Uma propriedade é um atributo associado a um valor, sendo definida como segue:

```
data Prop w
= forall a . (:=) (Attr w a) a
 | forall a . (:~) (Attr w a) (a    a)
 | forall a . (::=) (Attr w a) (w    a)
 | forall a . (::~) (Attr w a) (w    a    a)
```

Observe que existem quatro construtores para o tipo.

- **:=** Associa um valor a um atributo, ou seja, constrói a propriedade a partir de um atributo e de seu respectivo valor. Ex: text := “Haskell”
- **:** Constrói a propriedade a partir de uma função de atualização. Isto é, o valor da propriedade é obtido pela aplicação de uma função ao valor atual. Ex: text :~ (++”_Op”)
- **::=** Associa um atributo a um valor, com o uso de uma função que leva em conta o widget que estamos modificando.
- **::** Atualiza o valor de uma propriedade com uso de uma função que leva em conta o widget que estamos modificando

Podemos ler e modificar as propriedades de um objeto de interface por meio das funções get, set e swap:

- get :: w Attr w a IO a: Recebe um widget e um atributo deste widget, ao qual se deseja saber o valor associado, e retorna o valor da propriedade. Ex.: get w text retorna um valor do tipo IO (String)

- set :: w [Prop w] IO (): Atribui uma lista de propriedades ao *widget*. Ex.: set w [text := "WxHaskell", color := red,...].

- swap :: w Attr w a a IO a: Troca o valor de uma propriedade por outro e retorna o valor antigo.

Podemos ainda mapear um atributo de tipo a para um tipo b, por meio das funções:

- mapAttr :: (a b) (a b a) Attr w a Attr a b: Onde a função (a b) é usada para obter o valor da propriedade e a função (a b a) é usada para determinar o valor da propriedade.

- mapAttrW :: (v w) Attr v a Attr w a: Onde a função (v w) é usada para converter o objeto do tipo v em um objeto do tipo w.

Dissemos anteriormente que ações que criam controles têm como argumento listas de propriedades aplicáveis a esses controles. Para fins de organização, atributos estão definidos em classes, juntamente com as funções que os manipulam (se existirem). Veja, por exemplo, a classe Textual:

```
class Textual w where
  text :: Attr w String
  appendText :: w  String  IO ()
```

Esta classe define o tipo do atributo text e é instanciada para todos os *widgets* que possuem um *label*, ou um campo de texto. Nesta classe é também definido o tipo da função appendText que, dado um *widget* e uma *string*, acrescenta a *string* no final do texto que estava no *widget*.

Os atributos que podem ou não ser especificados para um dado objeto dependem das classes das quais esse objeto é instância.

Outras funções para manipulação de atributos e propriedades (incluindo a criação de novos atributos) estão definidas: veja o item atributos da seção wx da documentação da biblioteca *WxHaskell* para maiores informações.

1.4.1 – Eventos e tratadores de eventos

O evento é um tipo especial de atributo. Em geral, o tipo de um evento é: `Event w a`. Isto representa um evento para um *widget* `w` que espera um tratador de evento do tipo `a`. Podemos transformar um evento em um atributo com o uso da função on:

```
on :: Event w a  Attr w a
```

Em geral, associamos a um atributo evento uma ação `IO ()`, que é disparada quando o evento ocorre. Assim como os atributos, os eventos também são divididos em classes.
As classes `Commanding`, `Selecting` e `Reactive` compreendem classes de eventos básicos. A primeira define o evento

```
command :: Commanding w => Event w (IO ())
```

um tipo de evento que é disparado quando uma ação padrão acontece, como, por exemplo, um click em um botão ou o pressionamento da tecla enter em um campo de texto.

A classe `Selecting` define o evento `select`, que ocorre sempre que um item é selecionado de uma lista de itens em objetos como `listBox`, `choice` etc.

Já a classe `Reactive` (classe dos objetos visíveis) define eventos de resposta, como, por exemplo, o evento `resize`, que ocorre quando alteramos o tamanho de um *widget*, e o evento `closing`, que ocorre quando o objeto é fechado. É a esta classe que pertencem os eventos de mouse e de teclado.

1.4.1.1 – Eventos de mouse

O evento `mouse` tem o tipo

```
mouse :: Reactive w => Event w (EventMouse  IO () .
```

A definição do tipo do tratador de evento – `EventMouse` – é mostrada a seguir:

```
data EventMouse  =
        MouseMotion !Point !Modifiers
        MouseEnter !Point !Modifiers
        MouseLeave !Point !Modifiers
        MouseLeftDown !Point !Modifiers
        MouseLeftUp !Point !Modifiers
        MouseLeftDClick !Point !Modifiers
        MouseLeftDrag !Point !Modifiers
        MouseRightDown !Point !Modifiers
        MouseRightUp !Point !Modifiers
        MouseRightDClick !Point !Modifiers
        MouseRightDrag !Point !Modifiers
        MouseMiddleDown !Point !Modifiers
        MouseMiddleUp !Point !Modifiers
        MouseMiddleDClick !Point !Modifiers
        MouseMiddleDrag !Point !Modifiers
        MouseWheel !Bool !Point !Modifiers
```

Todos os eventos de mouse têm em comum o ponto onde o evento ocorreu (especificado em termos de coordenadas x e y) e os modificadores do evento – teclas como alt, shift, ctrl, que podiam estar pressionadas quando o evento ocorreu. O exemplo a seguir ilustra como usar o evento mouse. A função trataEvento1 trata o evento de o usuário "pressionar" o botão esquerdo do mouse. Essa função tem como argumentos a janela principal – apenas para mostrar um diálogo – e um EventMouse.

```
gui :: IO ()
gui = do f ← frame [text := "Exemplos: Eventos e Tratadores de eventos"]
         p ← panel f []
         b1 ← button p [text := "Click Here", on mouse := trataEvento1 f]
         set p [layout := (floatCentre.widget) b1]
         set f [layout := margin 5 (widget p)]

  where
    trataEvento1 w (MouseLeftDown p m) = infoDialog w "Evento de mouse"
                                                      "b1 down"
    trataEvento1 w _ = return ()
```

A função tratadora de evento captura, por casamento de padrão, o tipo de EventMouse ocorrido. No caso de ser pressionado do botão esquerdo do mouse, o EventMouse gerado casa com o padrão especificado na primeira equação de

definição da função trataEvento1 e o código correspondente é executado, exibindo na tela uma janela de um diálogo. A função infoDialog tem comportamento similar ao da função *JOptionPane.showMessageDialog* de Java. O tipo desta função é:

```
infoDialog :: Window a   String   String   IO ()
```

onde o primeiro argumento é a janela pai do diálogo, o segundo argumento é o título e o terceiro é a mensagem a ser mostrada.

Por conveniência os eventos de mouse e de teclado têm uma série de filtros que facilitam a descrição de tratadores de eventos. Os filtros para os eventos de mouse são:

```
enter         :: Reactive w => Event w (Point   IO ())
leave         :: Reactive w => Event w (Point   IO ())
motion        :: Reactive w => Event w (Point   IO ())
drag          :: Reactive w => Event w (Point   IO ())
click         :: Reactive w => Event w (Point   IO ())
unclick       :: Reactive w => Event w (Point   IO ())
doubleClick   :: Reactive w => Event w (Point   IO ())
clickRight    :: Reactive w => Event w (Point   IO ())
unclickRight  :: Reactive w => Event w (Point   IO ())
```

Todos os filtros funcionam simultaneamente e de forma independente. Sempre que o evento mouse for definido, todos os filtros são sobrescritos. Se for necessário redefinir o evento mouse, mas manter os antigos filtros, deve-se ler o evento mouse e chamá-lo novamente no novo *handler*:

```
(...)
set w [on mouse :~ \prev   do {...; prev}]
(...)
```

Em geral, cada filtro e/ou evento tem um tratador *default*. Quando definimos um tratador de evento, sobrescrevemos o tratador *default* – de modo análogo ao que ocorre em Java.

Quando um determinado evento não é tratado em um tratador de evento, deve ser chamada a função propagateEvent, para propagar o evento para o próximo tratador. O exemplo a seguir pode ajudar a entender melhor esse mecanismo de propagação de eventos:

```
main = start gui

gui = do f ← frame [text := "Propagação de eventos",
```

```
                        on closing ::= closer]
        b ← button f [text := "Fechar", on command := close f]

        set f [layout := margin 5 ((floatCentre.widget) b)]

 where closer w =  infoDialog w "Aviso" "Atenção este programa será fechado"
```

Execute o programa acima e veja o que acontece. Para fechar a janela, digite **crtl+c** na janela do GHCI. A janela não fecha porque o tratador de eventos apresentado acima não processa esse evento. Mas, se propagarmos esse evento, ele será processado pelo próximo *event handler* (geralmente no objeto pai) que, neste caso, será o evento closing *default* de Window. Para que a janela seja fechada corretamente, a função closer deve então ser reescrita do seguinte modo:

```
[...]
  where closer w = do infoDialog w "Aviso" "Atenção este programa será fechado"
                      propagateEvent
```

1.4.1.2 – Eventos de teclado

Tal como acontece com os eventos de mouse, temos um evento keyboard, que representa eventos do teclado, com o seguinte tipo:

```
keyboard :: Reactive w => Event w (EventKey    IO ())
```

onde EventKey é definido como:

```
data EventKey =
  EventKey !Key !Modifiers !Point

Instances
  Eq EventKey
  Show EventKey
```

onde Key representa uma tecla, Modifiers representas teclas modificadoras (como shift, alt e ctrl), que podem ter sido pressionadas quando o evento ocorreu, e Point representa o ponto de foco (coordenadas do cursor do mouse, em relação às coordenadas do *widget*, quando o evento ocorreu.).

Os eventos de teclado também possuem filtros:

```
anyKey      :: Reactive w => Event w (Key    IO ())
key         :: Reactive w => Key    Event w (IO ())
charKey     :: Reactive w => Char    Event w (IO ())
enterKey    :: Reactive w => Event w (IO ())
tabKey      :: Reactive w => Event w (IO ())
escKey      :: Reactive w => Event w (IO ())
```

```
helpKey      :: Reactive w => Event w (IO ())
delKey       :: Reactive w => Event w (IO ())
homeKey      :: Reactive w => Event w (IO ())
endKey       :: Reactive w => Event w (IO ())
pgupKey      :: Reactive w => Event w (IO ())
pgdownKey    :: Reactive w => Event w (IO ())
downKey      :: Reactive w => Event w (IO ())
upKey        :: Reactive w => Event w (IO ())
leftKey      :: Reactive w => Event w (IO ())
rightKey     :: Reactive w => Event w (IO ())
rebind       :: Event w (IO ())   Event w (IO ())
```

O que foi dito anteriormente sobre filtros de mouse também vale para filtros de teclado. O exemplo a seguir mostra a utilização de eventos do teclado, para filtrar uma entrada de texto de modo que apenas números inteiros positivos possam ser digitados:

```
main = start gui

gui = do f ← frame [text := "Entrada de texto com Filtros"]
         e ← entry f [on anyKey := handler]

         set f [layout := row 2 [label "Digite um número",widget e] ]

 where handler (KeyChar ch) =  if ((ch >= '0') && (ch < '9')) -- caractere de 0-9
                                  then propagateEvent     -- propaga evento
                                  else return ()          -- "barra"  evento

       handler (KeySpace) = return ()  -- trata separamente espaços em branco

       handler _ = propagateEvent      -- propaga outros eventos, como teclas de
                                       -- edição
```

Modifique o filtro acima de maneira que também seja possível digitar números com casa decimais.

1.4.1.3 – Outros eventos:

Além dos eventos de teclado e de mouse, a classe Reactive também define o tipo de outros eventos, sendo os principais listados a seguir:

closing :: Reactive w => Event w (IO ()) : Ocorre sempre que uma janela é fechada.

resize :: Reactive w => Event w (IO ()) : Ocorre quando uma janela é redimensionada

focus :: Reactive w => Event w (Bool IO ()) : Ocorre quando o widget ganha o foco.

activate :: Reactive w => Event w (Bool IO ()) : Ocorre quando a janela é ativada.

1.5 – Layouts

A biblioteca *WxHaskell* provê o tipo Layout para a representação de *layouts*, assim como funções (ou combinadores, como são chamados pelo autor da biblioteca) para criar e estruturar *layouts* a partir de *widgets*. O *layout* mais simples possível é definido pela função

```
widget :: Widget w => w  Layout
```

que cria um *layout* a partir de um *widget* qualquer.

Podemos ainda agrupar *layouts* em *containeres*, sendo os mais comuns os seguintes:

row :: Int [Layout] Layout : distribuição horizontal, com certo espaço (em pixels) entre os componentes

column :: Int [Layout] Layout : distribuição vertical, com certo espaço entre os componentes

grid :: Int Int [[Layout]] Layout : Uma distribuição em grade dos *layouts*, com um certo espaço separando os componentes verticalmente e horizontalmente.

Veja o exemplo a seguir:

```
import Graphics.UI.WX

main = start gui

        -- Cria 6 botões
gui = do f ← frame [text := "Layout Demo"] -- Cria a janela principal
        b1 ← button f [text := "1"]
        b2 ← button f [text := "2"]
```

```
        b3 ← button f [text := "3"]
        b4 ← button f [text := "4"]
        b5 ← button f [text := "5"]
        b6 ← button f [text := "6"]
        -- Seta o layout da janela principal para row (linha)
        set f [layout := row 2 (map widget [b1,b2,b3,b4,b5,b6])]
```

Como resultado da execução deste código, a seguinte janela será exibida:

Figura 3 - Layout em Linha (ROW)

Observe, no exemplo anterior, que, antes de usarmos a função row, convertemos cada um dos elementos da lista em *layout* (usando map widget), para só então transformar a lista de *layouts* resultante em um *layout row*.

Veja agora como ficaria o exemplo anterior, usando o *layout grid*:

```
import Graphics.UI.WX

main = start gui

gui = do f ← frame [text := "Layout Demo"] -- Cria a janela principal
         -- Cria 6 botões
         b1 ← button f [text := "1"]
         b2 ← button f [text := "2"]
         b3 ← button f [text := "3"]
         b4 ←button f [text := "4"]
         b5 ← button f [text := "5"]
         b6 ← button f [text := "6"]
         -- Seta o layout da janela principal para grid
         set f [layout := grid 2 2 (layoutFrom [ [b1,b2,b3], [b4,b5,b6] ])]
 where
   layoutFrom [] = []
   layoutFrom (x:xs) = map widget x:layoutFrom xs
```

A execução deste programa resulta na exibição da seguinte janela:

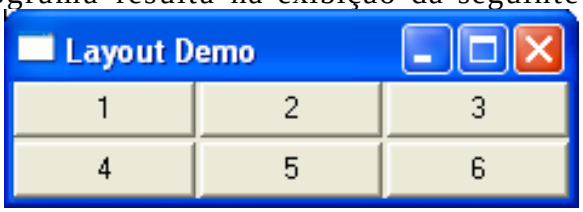

Figura 4 - Layout em grade

Uma observação importante sobre esses *containers* é que, por *default*, eles são estáticos, isto é, não acompanham o redimensionamento da janela. Se a intenção for que isso aconteça, é necessário tornar o *layout* extensível, usando o combinador

```
stretch :: Layout  Layout.
```

Os *containers* row e column, entretanto, só serão extensíveis se todos os seus elementos forem extensíveis. No caso do *container* grid, cada linha ou coluna será extensível se todos os elementos dessa linha ou coluna forem extensíveis. Vamos então substituir a ocorrência da função widget pela função composta stretch.widget, no corpo da função layoutFrom.

Execute novamente o programa e observe que agora o *layout* se expande quando aumentamos a janela, mas os botões apenas se realinham na janela. Para que os botões se expandam, encobrindo toda a área da janela, devemos usar o combinador

```
expand :: Layout  Layout.
```

Nossa função de *layout* ficaria então do seguinte modo: expand.stretch.widget. Isto é equivalente a usar o combinador fill, (fill.widget).

A função label :: String Layout permite que legendas sejam criadas diretamente como *layouts*. Isso é muito útil quando usamos legendas que não são alteradas em tempo de execução, como no exemplo a seguir:

```
import Graphics.UI.WX

main = start gui

gui = do f ← frame [text := "Layout demo 2 – labels"]
         name ← entry f []
         tel ← entry f []
         set f [layout := grid 2 2 [ [label "Nome:", widget name],
                                     [label "Telefone:", widget tel] ] ]
```

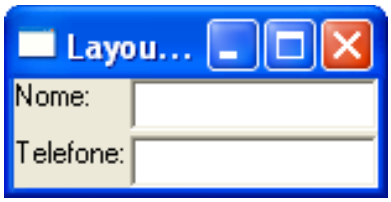

Figura 5 - Utilização de labels

É possível adicionar margens (margin :: Int Layout Layout) e *boxes* (boxed :: String Layout Layout) a um determinado *layout*. A margem pode ser adicionada a todo o *layout*, ou individualmente, à direita (marginRight), à esquerda (marginLeft), ao cabeçalho (marginTop) ou ao roda-pé (marginBottom). A seguir, modificamos o exemplo anterior, para adicionar um *box* e uma margem de cinco pixels.

```
import Graphics.UI.WX

main = start gui

gui = do f ← frame [text := "Layout demo 2 – labels"]
         name ← entry f []
         tel ← entry f []
         set f [layout := layoutAdapter (grid 2 2 [[label "Nome:",widget name],
                                                   [label "Telefone:", widget tel]]) ]

 where layoutAdapter = (margin 5).(boxed "Dados pessoais")
```

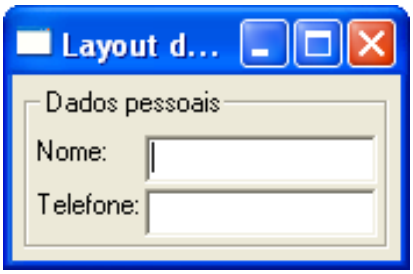

Figura 6 - Janela com margem e box

É possível, ainda, criar um *layout* "vazio". A função empty:: Layout retorna um layout vazio, extensível em todas as direções. Existem ainda as funções:

hspace	:: Int Layout : espaço horizontal com certo comprimento
vspace	:: Int Layout : espaço vertical com uma certa altura
space	:: Int Int Layout : espaço com um comprimento e uma altura

Por fim, existem também funções para alinhamento, que são:

centre	:: Layout Layout : alinha o layout ao centro
alignTopLeft	:: Layout Layout : alinha o layout ao canto superior esquerdo
alignTop	:: Layout Layout : alinha o layout ao cabeçalho
alignTopRight	:: Layout Layout : alinha o layout ao canto superior direito
alignLeft	:: Layout Layout : alinha o layout à esquerda
alignCentre	:: Layout Layout : alinha o layout ao centro
alignRight	:: Layout Layout : alinha o layout à direita

alignBottomLeft	:: Layout	Layout : alinha o layout ao canto inferior esquerdo
alignBottom	:: Layout	Layout : alinha o layout ao roda- pé
alignBottomRight	:: Layout	Layout : alinha o layout ao canto inferir direito

Para cada uma dessas funções existe também um versão que torna o *layout* extensível. Isto é, à função alignCentre corresponde a função floatCentre e assim por diante.

Layouts mais complexos podem ser descritos separando o *layout* em partes, e construindo cada parte um container separado (geralmente um panel). Em seguida, usamos um *layout* conveniente para agrupar cada uma das partes. Veja o exemplo a seguir:

```
module LayoutDemo3 where

import Graphics.UI.WX

main = start gui
gui = do f ← frame [text := "LayoutDemo 3"]
         p1 ← panel f []
         p2 ← panel f []
         p3 ← panel f []

         nome ← entry p1 []
         btAdd ← button p1 [text := "adicionar"]
         set p1 [layout :=  row 2 [label "Nome: ",widget nome, widget btAdd] ]

         w1 ← entry p2 [clientSize := sz 70 5]
         w2 ← entry p2 [text := "OK", clientSize := sz 70 20]
         w3 ← hgauge p2 100 [selection := 50,clientSize := sz 70 10]
         w4 ← button p2 [text := "Click-me", on command := set w3 [selection :~(+1)]]
         w5 ← checkBox p2 [text := "Haskell", checked := True,
                           on command ::= (\ch -> set ch [text :~ reverse]) ]
         w6 ← button p2 [bgcolor := cyan]
         set p2 [layout := grid 6 6 [[widget w1,widget w3,widget w4],
                                     [widget w6,widget w5,widget w2] ]  ]

         btSair ← button p3 [text := "Sair", on command := close f]
         set p3 [layout := (floatBottomRight.widget) btSair]
         set f [layout := margin 5 (column 3 [((boxed "").fill.widget) p1,
                                              ((boxed "").fill.widget) p2,
                                              (fill.widget) p3]) ]
```

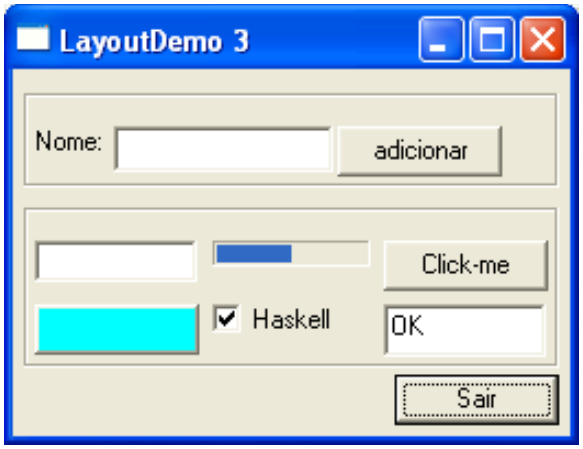

Figura 7 - Usando panels para criar layouts mais elaborados

Neste exemplo, dividimos o *layout* em três panels, p1, p2 e p3. Em p1 temos um *layout* em coluna, em p2 um *layout grid* e em p3 temos um botão alinhado à direita.

1.6 – Variáveis Mutáveis:

Algumas vezes, podemos querer armazenar um valor de algum tipo, de modo que ele possa ser modificado posteriormente. Geralmente, tal modificação envolve a reconstrução do tipo com um novo valor. A utilização de variáveis mutáveis é uma maneira simples de lidar com estas situações. Elas permitem que, em uma mônada do tipo IO (), um determinado valor seja "armazenado" e posteriormente modificado.

As operações definidas para variáveis simples são:

varCreate	:: a IO (Var a) : Cria uma variável mutável para armazenar valores do tipo a.
varGet	:: Var a IO a: Obtém o valor da variável.
varSet	:: Var a a -> IO () : Define um valor para a variável.
varUpdate	:: Var a (a a) IO a : Altera o valor da variável e retorna o valor antigo.
varSwap	:: Var a a IO a : Troca o valor da variável e retorna o valor antigo.

2 – Controles de interface gráfica.

Nesta seção, descrevemos, de maneira um pouco mais detalhada, os principais controles e *containeres* implementados na biblioteca, bem como os principais atributos destes controles. Para tal, vamos nos concentrar nos seguintes tópicos:

- **Criação:** Como criar o objeto.
- **Styles:** Estilos aceitos pelo objeto (se houverem) – um conjunto de constantes que permitem modificar o comportamento de alguns controles da interface gráfica. Para combinar estas constantes utiliza-se o operador `(.+.) :: Int -> Int -> Int` (bitwise or), definido no módulo *WXCore*.
- **Funções utilitárias:** Funções especiais usadas para manipular o objeto.
- **Instâncias e atributos relevantes:** Classes instanciadas e/ou atributos do objeto.
- **Lista de instâncias:** Lista das demais classes das quais o objeto é instância.

2.1 – Window

O `Window` é a origem de praticamente todos os demais *widgets* visíveis. Por isso, os atributos presentes neste objeto também estão presentes na maioria dos demais controles de interface gráfica.

Criação:

- `Window a  [Prop (Window ())]  IO (Window ())`: Cria uma janela padrão.

Styles:

Nome da constante	Descrição
wxBORDER	Exibe uma borda simples ao redor da janela.
wxDOUBLE_BORDER	Exibe uma borda dupla ao redor da janela (apenas Windows)
wxSUNKEN_BORDER	Exibe uma borda rebaixada ao redor da janela.
wxRAISED_BORDER	Exibe uma borda em alto relevo. (apenas GTK)
wxSTATIC_BORDER	Exibe uma borda justa para um controle estático (apenas Windows)
wxTRANSPARENT_WINDOW	Define a janela como transparente, i.e., ela não receberá nenhum evento `paint`. (apenas Windows)
wxNO_3D	Desabilita efeitos 3D dos controles alojados nesta janela. (apenas Windows)

wxTAB_TRAVERSAL	Use este *flag* para impedir que a janela reaja a eventos *tab*.
wxWANTS_CHARS	Use isto para fazer com que a janela processe todos os eventos de teclado.
wxNO_FULL_REPAINT_ON_RESIZE	Impede que a janela seja completamente redesenhada quando redimensionada. (apenas Windows)
wxVSCROLL	Adiciona uma barra de rolagem vertical à janela
wxHSCROLL	Adiciona uma barra de rolagem horizontal janela
wxCLIP_CHILDREN	Utilize este estilo para corrigir distorções causadas pelo evento paint do plano de fundo. (apenas Windows)

Funções utilitárias:

- refit :: Window a → IO () : Assegura que o objeto esteja inteiramente contido dentro da área da janela quando o tamanho do objeto muda.

- refitMinimal :: Window a → IO (): Análoga à função anterior, mantendo a janela no tamanho mínimo aceitável.

Instâncias e atributos relevantes:

- rootParent :: ReadAttr (Window a) (Window ()) : Atributo que fornece a janela que é a raiz do *widget*.
- frameParent :: ReadAttr (Window a) (Window ()): Atributo que fornece o controle pai do *widget*.
- tabTraversal :: Attr (Window a) Bool: Atributo que controla o processamento dos *tabs*. Se desabilitado (False) a janela não processa os *tabs*.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.2 - Frames

Criação:

- frame :: [Prop (Frame ())] → IO (Frame ()): Cria frame padrão.
- frameFixed :: [Prop (Frame ())] → IO (Frame ()): Cria frame não redimensionável.

- frameTool :: [Prop (Frame ())] Window a IO (Frame ()): Cria um frame com estilo de caixa de ferramentas (flutua sobre a janela pai e possui uma barra de título)
- frameEx :: Style [Prop (Frame ())] Window a IO (Frame ()) : Cria frame com um determinado estilo.

Styles:

Nome da constante	Descrição
wxDEFAULT_FRAME_STYLE	Estilo padrão de janela: redimensionável, com barra de título, botões maximizar, minimizar etc.
wxICONIZE	Mostra a janela minimizada. (apenas Windows)
wxCAPTION	Mostra a barra de titulo.
wxMINIMIZE_BOX	Mostra o botão de minimizar
wxMAXIMIZE	Mostra a janela maximizada (apenas Windows)
wxMAXIMIZE_BOX	Mostra o botão de minimizar
wxSTAY_ON_TOP	Faz com que a janela fique sobre todas as outras janelas. (apenas Windows)
wxSYSTEM_MENU	Mostra um menu de sistema, geralmente à esquerda do título.
wxNO_BORDER	Não mostra bordas ou decorações (apenas Windows e GTK)
wxBORDER	Mostra uma borda redimensionável ao redor da janela.
wxFRAME_TOOL_WINDOW	Faz com que a barra de títulos fique pequena. Um frame com este estilo, não aparecerá na barra de tarefas no Windows.
wxFRAME_FLOAT_ON_PARENT	Faz com que o frame fique sempre no topo da janela pai.
wxFRAME_SHAPED	Permite que a forma da janela seja alterada.

O estilo padrão para o frame é definido como a soma das seguintes constantes:
wxMINIMIZE_BOX + wxMAXIMIZE_BOX + wxRESIZE_BORDER + wxSYSTEM_MENU + wxCAPTION

Observe que algumas das funções anteriores (como frameFixed, por exemplo) podem ser definidas usando a função frame e um conjunto adequado de *styles*.

Funções utilitárias:

As seguintes funções são exportadas pelo módulo *WXCore*:

- frameCreateTopFrame :: String IO (Frame ()): Cria um frame padrão e faz com que ele esteja sempre no topo das outras janelas.
- frameCreateDefault :: String IO (Frame ()): Cria um frame padrão.
- frameSetTopFrame :: Frame a IO (): Faz com o frame esteja sempre no topo das outras janelas.
- frameDefaultStyle :: Int: Especifica o estilo padrão para um frame.
- frameCenter :: Frame a IO (): Centraliza o frame na tela.
- frameCenterHorizontal :: Frame a IO (): Centraliza o frame horizontalmente
- frameCenterVertical :: Frame a IO (): Centraliza o frame verticalmente.

Outras funções utilitários para Window:

- windowGetRootParent :: Window a IO (Window ()) : Retorna a raiz do widget.
- windowGetFrameParent :: Window a IO (Window ()): Retorna a janela ou diálogo pai de um widget.
- windowGetMousePosition :: Window a IO Point: Retorna a posição atual do mouse relativa à posição da janela.
- windowGetScreenPosition :: Window a IO Point : Retorna a posição da janela relativa à origem da tela.
- windowChildren :: Window a IO [Window ()] : Retorna todos os widgets filhos da janela.

Instâncias e atributos relevantes:

As classes Form, Closable e Framed são essenciais para este objeto. A classe Form determina o atributo Layout, como vimos anteriormente. A classe Closeable especifica o método close:: Window a IO (), que possibilita que o frame seja fechado. A classe Framed determina os atributos, que devem ser atribuídos ao objeto, quando este é criado:

- resizeable :: Attr w Bool : Determina se o widget pode ser redimensionado pelo usuário.
- minimizeable :: Attr w Bool : Determina se o widget pode ser minimizado pelo usuário.
- maximizeable :: Attr w Bool : Determina se o widget pode ser maximizado pelo usuário.
- closeable :: Attr w Bool: Determina se o widget pode ser fechado pelo usuário.

Outros atributos relevantes são:

- menuBar :: WriteAttr (Frame a) [Menu ()] : Atributo somente de escrita, que especifica a barra de menu associada ao frame.
- statusBar :: WriteAttr (Frame a) [StatusField]: Atributo somente escrita, que especifica a statusBar associada ao frame.

Lista de instâncias: O frame também instancia as seguintes classes: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.3 – MDI Frames

MDI são *frames* que permitem a exibição de vários *frames* em seu interior. Geralmente, temos um MDIParentFrame que serve de *container* para MDIChildFrames.

Criação:

- mdiParentFrame :: [Prop (MDIParentFrame ())] IO (MDIParentFrame ()) : Cria um MDIParentFrame.
- mdiChildFrame :: MDIParentFrame a [Prop (MDIChildFrame ())] IO (MDIChildFrame ()): Cria um MDIChildFrame.
- mdiParentFrameEx :: Window a -> Style [Prop (MDIParentFrame ())] IO (MDIParentFrame ()) : Cria o parentFrame com um determinado estilo.
- mdiChildFrameEx :: MDIParentFrame a Style [Prop (MDIChildFrame ())] IO (MDIChildFrame ()) : Cria um childFrame com um determinado estilo.

Styles:

Para o MDIParentFrame temos, além dos estilos básicos de frame (wxICONIZE, wxCAPTION, wxDEFAULT_FRAME_STYLE, wxMINIMIZE_BOX, xMAXIMIZE, wxMAXIMIZE_BOX, wxSTAY_ON_TOP, wxSYSTEM_MENU), os seguintes estilos de forma adicionais:

Nome da constante	Descrição
wxHSCROLL	Exibe uma barra de rolagem horizontal na área cliente.
wxVSCROLL	Exibe uma barra de rolagem vertical na área cliente.
wxRESIZE_BORDER	Mostra uma borda redimensionável ao redor da janela (apenas Motif).
wxFRAME_NO_WINDOW_MENU	Em Windows, remove o menu de sistema que é automaticamente adicionado.

Para o MDIChildFrame, os estilos possíveis são os estilos básicos de frame (relacionados no parágrafo acima), e mais os seguintes: wxRESIZE_BORDER, wxSTAY_ON_TOP, wxSYSTEM_MENU e wxTHICK_FRAME.

Funções utilitárias:

- activeChild :: ReadAttr (MDIParentFrame a) (MDIChildFrame ()): Atributo de leitura, que informa o MDIChild ativo (objectIsNull, se não houver nenhuma janela ativa).
- activateNext :: MDIParentFrame a IO () : Ativa o próximo child frame.
- activatePrevious :: MDIParentFrame a IO (): Ativa o ChildFrame Anterior
- arrangeIcons :: MDIParentFrame a IO (): Organiza os ChildFrames minimizados.
- cascade :: MDIParentFrame a -> IO (): Arranja os ChildFrames em cascata.
- tile :: MDIParentFrame a -> IO () : Arranja os ChildFrames lado a lado.

Instâncias e atributos relevantes: As mesmas mencionadas para frame.

Lista de instâncias: Mesma lista de instâncias de frame.

2.4 - Panel

Um Panel é geralmente usado como um *container* para outros objetos.

Criação:

- panel :: Window a [Prop (Panel ())] IO (Panel ()): Cria um panel padrão.
- panelEx :: Window a Style [Prop (Panel ())] IO (Panel ()): Cria um panel com um determinado estilo.

Styles:

Não há nenhum estilo específico para este objeto.

Funções utilitárias:

- focusOn :: Window a IO () : Seta o foco inicial

Instâncias e atributos relevantes:

Destacamos a classe Form coma a única instância relevante para o Panel. O atributo defaultButton permite especificar um botão padrão, i. e., sempre que a tecla *enter* e o Panel estiverem em foco, o evento command do botão padrão é disparado.

Lista de instâncias: Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Textual, Literate, Reactive e Paint.

2.5 – Notebook

Assim como o Panel, o Notebook também é utilizado como *container* para controles. A diferença é que no Notebook podemos organizar vários *containeres* em abas.

Criação:

- notebook :: Window a [Prop (Notebook ())] IO (Notebook ())

Styles:

Nome da constante	Descrição
wxNB_LEFT	Tabs são colocadas à esquerda (não suportado em Windows xp)
wxNB_RIGHT	Tabs são colocadas à direita. (não suportado em Windows xp)
wxNB_BOTTOM	Tabs são colocadas na parte inferior. (não suportado em Windows xp)
wxNB_FIXEDWIDTH	Todas as tabs têm o mesmo comprimento (apenas Windows).
wxNB_MULTILINE	Podem haver várias linhas em uma tab. (apenas Windows)

Funções utilitárias:

As seguintes funções são usadas para controlar o Layout:

- tab :: String Layout TabPage: Cria uma página simples com tipo e layout especificados.
- imageTab :: String Bitmap () Layout TabPage: Cria uma página com título, layout e ícone especificados.
- tabs :: Notebook a [TabPage] Layout : Cria o layout Notebook.

Todas as TabPages devem sempre estar contidas em um *container* (geralmente, um Panel)

Instâncias e atributos relevantes: Nenhum atributo relevante em particular.

Lista de instâncias: Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Textual, Literate, Reactive, Paint

2.6 – Button

Criação:

- button :: Window a [Prop (Button ())] IO (Button ()): Cria um simples botão padrão.
- buttonEx :: Window a Style [Prop (Button ())] IO (Button ()): Cria um botão com estilo especificado.
- smallButton :: Window a [Prop (Button ())] IO (Button ()): Cria um botão de tamanho mínimo.
- bitmapButton :: Window a [Prop (BitmapButton ())] IO (BitmapButton ()): Cria um botão que pode possui uma imagem associada.

Styles:

Nome da constante	Descrição
wxBU_LEFT	Alinhado à esquerda. (WIN32 apenas).
wxBU_TOP	Alinha o texto ao topo do botão.(WIN32 apenas).
wxBU_RIGHT	Alinhado à direita. (WIN32 apenas).
wxBU_BOTTOM	Alinha o texto ao rodapé do botão. (WIN32 apenas).
wxBU_EXACTFIT	Cria um botão tão pequeno quanto possível. (procedimento padrão)

Funções utilitárias: Nenhuma função utilitária.

Instâncias e atributos relevantes:

Commanding: Define o evento (command). Para o bitmapButtom, o atributo picture :: String informa o caminho para o ícone.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.7 – TextCtrl

Criação:

- entry :: Window a [Prop (TextCtrl ())] IO (TextCtrl ()): Cria uma entrada de texto de apenas uma linha.
- textEntry :: Window a [Prop (TextCtrl ())] IO (TextCtrl ()):Cria uma entrada de texto de apenas uma linha.

- textCtrl :: Window a [Prop (TextCtrl ())] IO (TextCtrl ()): Cria uma entrada de texto de várias linhas .
- textCtrlRich :: Window a [Prop (TextCtrl ())] IO (TextCtrl ()): Cria uma entrada de texto de várias linhas que possibilita a utilização de formatação específica em partes do texto (em geral só funciona no Windows).
- textCtrlEx :: Window a Style [Prop (TextCtrl ())] IO (TextCtrl ()): Cria uma entrada de texto com o estilo especificado.

OBS: Para as entradas de texto de uma única linha, devemos especificar o atributo alignment quando o objeto for criado. Para entradas de texto de múltiplas linhas, além do alignment, também deve ser especificado o atributo warp.

Styles:

Nome da constante	Descrição
wxTE_PROCESS_ENTER	Faz com que o controle gere a mensagem wxEVENT_TYPE_TEXT_ENTER_COMMAND. Caso contrário, o evento é processado internamente pelo controle.
wxTE_PROCESS_TAB	Faz com que o controle capture Tabs. Geralmente, Tab é usado para passar ao próximo controle da janela. Para controles criados com este estilo, pode ser usado *ctrl+enter* para passar ao próximo controle ativo da janela.
wxTE_MULTILINE	Permite múltiplas linhas.
wxTE_PASSWORD	O texto será ecoado como asteriscos
wxTE_READONLY	O texto não será editável pelo usuário.
wxTE_RICH	Usa *ricth text* em Win32. Isto habilita ao controle usar mais do que 64kb para texto. Este estilo é ignorado por outras plataformas.
wxTE_RICH2	Usa a versão 2.0 ou 3.0 do *rich text*. (Windows apenas)
wxTE_AUTO_URL	Destaca URL's. Este estilo é suportado apenas sobre Win32 e requer wxTE_RICH.
wxTE_NOHIDESEL	Por padrão, controles de texto Widnows não exibem seleção quando não têm o foco. Use este estilo para forçar o controle a exibir a seleção mesmo quando ele não tiver o foco. (Este estilo é ignorado em outras plataformas.)
wxHSCROLL	Adiciona um scrollbar horizontal. Sem efeitos

	em GTK.
wxTE_LEFT	O controle é alinhado à esquerda
wxTE_CENTRE	O controle é centralizado
wxTE_RIGHT	O controle é alinhado à direita.
wxTE_DONTWRAP	O mesmo que wxHSCROLL
wxTE_LINEWRAP	Dispõe a linha de modo que ela fique inteira.
wxTE_WORDWRAP	Dispõe a linha de modo que todas as palavras nela presentes fiquem inteiras.

Funções utilitárias:

- appendText :: w String IO (): Adiciona um texto ao controle.

Instâncias e atributos relevantes:

Atributos:

- processEnter :: Styled w => Attr w Bool: Processar evento de teclado <enter>. Este evento pode ser capturado com um oncommand.
- processTab :: Styled w => Attr w Bool : Processar evento de teclado <tab>.

Instâncias:

- Wrap : Define o modo de disposição da linha. Poder ser WrapNone, WrapLine ou WrapWord.
- Aligned : Define o alinhamento (alignment :: Aligned w => CreateAttr w Align). Pode ser AlignLeft, AlignRight ou AlignCentre.
- Commanding: Define o evento oncommand.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.9 – Check Box

Criação:

- checkBox :: Window a [Prop (CheckBox ())] IO(CheckBox ()): Cria uma checkBox padrão.

Styles:

Nenhum estilo em particular para este widget.

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

Instâncias:

- Checkable:
  - checkable :: Attr w Bool : Informa se o objeto está marcável, ou não
  - checked :: Attr w Bool : Informa se o objeto está marcado ou não.

- Commanding.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.10 – Choice

Criação:

- choice :: Window a [Prop (Choice ())] IO (Choice ()) : Cria uma lista de strings e permite a escolha de uma string.

Styles:

Nenhum estilo em particular para este *widget*.

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Sorted
  - sorted :: Sorted w => CreateAttr w Bool : Se este atributo estiver marcado como True, a lista de strings será classificada em ordem alfabética.
- Selecting :
  - select :: Selecting w => Event w (IO ()) : Evento que ocorre quando um determinado item é selecionado.
- Selection :
  - selection :: Attr w Int : Atributo que informa o índice (baseado em zero) da string selecionada.
- Items :
  - itemCount :: ReadAttr w Int : Atributo somente leitura que fornece a quantidade de itens armazenados no *widget*.

- items :: Attr w [a] : Informa todos os elementos em uma lista.(Este atributo pode não permitir escrita em determinados *widgets*)
- item :: Int -> Attr w a : Retorna um atributo correspondente ao item existente ma posição passada como argumento.
- itemDelete :: w Int IO () : Exclui o item da posição correspondente. (Válido apenas para o caso de itens com escrita)
- itemsDelete :: w IO () : Exclui todos os itens. (Válido apenas para o caso em de items com escrita)
- itemAppend :: w a IO () : Adiciona um item. (Válido apenas para o caso de itens com escrita)

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.11 – ComboBox

Criação:

- comboBox :: Window a [Prop (ComboBox ())] IO (ComboBox ()): Cria uma lista de seleção *combo* (uma caixa com uma única linha, sobre a qual se pode clicar para que seja exibida uma lista de itens para seleção).
- comboBoxEx :: Window a Style [Prop (ComboBox ())] IO (ComboBox ()): Cria uma comboBox com o estilo especificado.

Styles:

Nome da constante	Descrição
wxCB_SIMPLE	Cria uma comboBox com uma lista de itens em exibição permanente.
wxCB_DROPDOWN	Cria uma comboBox com uma lista drop- down.
wxCB_READONLY	O mesmo que wxCB_DROPDOWN, mas apenas strings que já tiverem sido previamente adicionadas ao comboBox poderão ser escolhidas. (Não permite edição)
wxCB_SORT	Ordena as entradas da lista alfabeticamente.

Funções utilitárias: Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Selecting

- Commanding
- Selection
- Items

OBS: Um evento command só poderá ser capturado se processEnter for True.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.12 – ListBox

Criação:

- singleListBox :: Window a [Prop (SingleListBox ())] IO (SingleListBox ()): Cria uma lista de seleção que permite seleção de apenas um item.
- multiListBox :: Window a [Prop (MultiListBox ())] IO (MultiListBox ()):Cria uma lista de seleção que permite seleção de mais de um item.

Styles:

Nome da constante	Descrição
wxLB_SINGLE	Lista de seleção simples (permite a seleção de apenas um item).
wxLB_MULTIPLE	Lista de multi- seleção (Permite a seleção de mais de um item).
wxLB_EXTENDED	Seleção estendia. O usuário pode selecionar múltiplos itens, usando a tecla Shift e outros combinadores.
wxLB_ALWAYS_SB	Sempre exibir barra de rolagem vertical.
wxLB_NEEDED_SB	Cria barra de rolagem vertical se necessário
wxLB_SORT	O conteúdo da lista será classificado em ordem alfabética.

OBS: wxLB_SINGLE, wxLB_MULTIPLE e wxLB_EXTENDED são mutuamente exclusivos.

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Sorted

- Selecting
- Selection
- Items

Obs : para o caso de multiListBox, existe uma instância relevante especial – Selections – que define o atributo selections :: Attr w [Int], o qual informa os elementos selecionados, se houver mais de um.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.13 – RadioBox

Criação:

- radioBox :: Window a Orientation [String] [Prop (RadioBox ())] IO (RadioBox ()): Cria um caixa de botões radio. Os botões são semelhantes ao checkBox, mas são mutuamente exclusivos.

Obs: O argumento Orientation determina a disposição dos itens. Os construtores são Horizontal e Vertical. A lista de strings passada como argumento refere-se aos *labels* de cada opção.

Styles:

Nome da constante	Descrição
wxRA_SPECIFY_ROWS	O parâmetro de maior dimensão refere-se ao número máximo de linhas.
wxRA_SPECIFY_COLS	O parâmetro de maior dimensão refere-se ao número máximo de colunas.

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Selecting
- Selection
- Items

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.14 – Spin Control

Criação:

- spinCtrl :: Window a Int Int [Prop (SpinCtrl ())] IO (SpinCtrl ()): Cria um *spinControl* (uma entrada de texto com botões up/down). O usuário pode selecionar um valor numérico (pelo atributo selection) que esteja entre os valores passados pelo primeiro e segundo argumentos.

Styles:

Nome da constante	Descrição
wxSP_ARROW_KEYS	O usuário pode usar teclas de direção.
wxSP_WRAP	Os valores máximo e mínimo estão disponíveis.

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Selection
- Selecting

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.15 – Slider

Criação:

- Hslider :: Window a Bool Int Int [Prop (Slider ())] IO (Slider ()): Cria um Slider horizontal. (Um Slider é um apontador que se desloca sobre uma régua). Especifique o argumento booleano como True, para exibir os valores máximo, mínimo e corrente.
- vslider :: Window a Bool Int -> Int [Prop (Slider ())] IO (Slider ()): Cria um Slider vertical. Especifique o argumento booleano como True, para exibir o valor máximo, mínimo e corrente.
- sliderEx :: Window a -> Int Int Style [Prop (Slider ())] IO (Slider ()): Cria um Slider com o estilo especificado. Utilize o atributo selection para obter o valor corrente.

Styles:

Nome da constante	Descrição
wxHORIZONTAL	Exibe o slider horizontalmente.
wxVERTICAL	Exibe o slider verticalmente.
wxSL_AUTOTICKS	Mostra marcação.
wxSL_LABELS	Exibe os valores mínimo, máximo e corrente. (Apenas exibe o valor corrente em wxGTK)
wxSL_LEFT	Se for um slider vertical, exibe a marcação à esquerda.
wxSL_RIGHT	Se for um slider vertical, exibe a marcação à direita.
wxSL_TOP	Exibe a marcação ao topo, se for um slider horizontal.
wxSL_SELRANGE	Permite que o usuário escolha os uma faixa de valores no slider. (Apenas Windows 95).

Funções utilitárias:

Nenhuma função utilitária em particular.

Instâncias e atributos relevantes:

- Commanding
- Selection

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.16 – Gauge

Criação:

- hgauge :: Window a Int [Prop (Gauge ())] IO (Gauge ()): Cria uma barra de progresso horizontal. O parâmetro inteiro é usado para determinar o valor máximo.
- vgauge :: Window a Int [Prop (Gauge ())] IO (Gauge ()): Cria uma barra de progresso vertical.
- gaugeEx :: Window a Int Style [Prop (Gauge ())] IO (Gauge ()): Cria uma barra de progresso com o estilo especificado fornecido.

Para determinar a posição da barra de progresso utilize o atributo selection.

Styles:

Nome da constante	Descrição
wxHORIZONTAL	Cria uma barra de progresso horizontal.
wxVERTICAL	Cria uma barra de progresso vertical.
wxGA_PROGRESSBAR	No Windows 95 cria uma barra de progresso horizontal.
wxGA_SMOOTH	No Windows 95 cria uma barra de progresso com o incremento de um pixel por passo.

Funções utilitárias:

Nenhuma função utilitária.

Instâncias e atributos relevantes:

- Selection

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.17 – Static text

Criação:

- staticText :: Window a [Prop (StaticText ())] IO (StaticText ()): Cria um tipo de *label* (*widget* cujo único propósito é conter um texto).

Styles:

Nome da constante	Descrição
wxALIGN_LEFT	Alinha o texto à esquerda.
wxALIGN_RIGHT	Alinha o texto à direita.
wxALIGN_CENTRE	Centraliza o texto (Horizontalmente)
wxST_NO_AUTORESIZE	Por padrão, o controle ajusta seu tamanho automaticamente para cobrir a área onde está exibido. Utilize este flag para impedir redimensionamentos automáticos.

Funções utilitárias:

Nenhuma função utilitária.

Instâncias e atributos relevantes:

- Textual

Lista de instâncias: Sem mais instâncias.

2.18 – Splitter Window

Criação:

- splitterWindow :: Window a [Prop (SplitterWindow ())] IO (SplitterWindow ()): Cria um divisor de janelas. Um divisor de janelas é um barra (horizontal ou vertical) que divide a janelas em duas sub-janelas. Após dividir a janela, cada sub-janela pode possuir seu próprio *layout*.

Para trabalhar com *layouts* em janelas dividas usamos os combinadores vsplit e hsplit.

Styles:

Nome da constante	Descrição
wxSP_3D	Desenha a borda em 3D.
wxSP_3DSASH	Desenha a moldura em 3D.
wxSP_3DBORDER	Desenha a borda em 3D.
wxSP_FULLSASH	Desenha o fim da SashBar (Sob Windows torna desnecessário o uso de bordas.).
wxSP_BORDER	Desenha uma borda negra fina ao redor da janela.
wxSP_NOBORDER	Sem borda, e moldura negra.
wxSP_PERMIT_UNSPLIT	Sempre permite unisplit, ajustando com o Panel de menor tamanho.
wxSP_LIVE_UPDATE	Não desenha a linha XOR, mas redimensiona as janelas filhas

	imediatamente.

Funções utilitárias:

- hsplit :: SplitterWindow a Int Int Layout Layout Layout: Adiciona uma barra horizontal entre duas janelas. Os dois argumentos inteiros especificam a espessura da barra separadora e a altura inicial do painel mais alto respectivamente. Os dois argumentos seguintes definem o *layout* da parte superior e o *layout* da parte inferior.
- vsplit :: SplitterWindow a Int Int Layout Layout Layout: Adiciona uma barra vertical entre duas janelas. Os dois argumentos inteiros especificam a espessura da barra separadora e o comprimento inicial da sub-janela esquerda. Os dois argumentos seguintes especificam o layout da parte direita e o layout da parte esquerda.

Instâncias e atributos relevantes:

Nenhuma instância e/ou atributos relevantes.

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.19 – Image List

O ImageList é um objeto cujo propósito é conter um conjunto de imagens que podem ser utilizadas na aplicação (por exemplo, em botões, caixas de seleção e outros controles). Trata-se de um controle não visual.

Criação:

- imageList :: Size IO (ImageList ()): Cria uma ImageList vazia para armazenar imagens do tamanho especificado.
- imageListFromFiles :: Size [FilePath] IO (ImageList ()): Cria uma ImageList contendo todas as imagens da lista de caminhos de arquivos de imagens. Todas as imagens da lista serão redimensionadas para o tamanho passado como argumento.

OBS: O tamanho pode ser obtido por meio da função sz :: Int -> Int -> Size, onde o primeiro argumento é o comprimento e o segundo é a altura.

Styles:

Nenhum estilo disponível.

Funções utilitárias:

Para utilizar as funções a seguir, é necessário importar o módulo *WXCore*.

- imageListAddBitmap :: ImageList a Bitmap b Bitmap c IO Int : Adiciona um objeto BitMap à lista de imagens . O objeto Bitmap b é a figura em si e o outro é uma máscara de bitmap (que pode ser, por exemplo, uma figura de plano de fundo).
- imageListAddMasked :: ImageList a Bitmap b Color IO Int: Adiciona um objeto BitMap à lista de imagens. A cor passada como argumento é usada como máscara.
- imageListAddIcon :: ImageList a Icon b IO Int: Adiciona um ícone à lista de imagens.
- imageListDraw :: ImageList a Int DC c Point Int Bool IO Bool: Tem como argumentos um índice de uma imagem do ImageList, um dispositivo gráfico (geralmente informado pelo evento paint), uma coordenada, um *flag* de desenho e um valor booleano que informa se o plano de fundo dever ser desenhado de maneira sólida. Desenha a imagem correspondente ao índice passado como argumento.
- imageListRemove :: ImageList a Int IO Bool : Remove o elemento correspondente ao índice passado como argumento.
- imageListRemoveAll :: ImageList a IO Bool: Esvazia o imageList.

Instâncias e atributos relevantes:

Nenhuma instância e/ou atributos relevantes.

Lista de instâncias: Nenhuma Instância.

2.20 – List Control

Criação:

- listCtrl :: Window a [Prop (ListCtrl ())] IO (ListCtrl ()): Cria um ListCtrl simples (em modo Report).
- listCtrlEx :: Window a Style [Prop (ListCtrl ())] IO (ListCtrl ()): Cria um ListCtrl com o estilo especificado.

Um ListCtrl é basicamente uma lista de strings. A principal diferença entre ele e o ListBox é que o primeiro permite mais de uma coluna.

Styles:

Nome da constante	Descrição

wxLC_LIST	Lista de visualização em multicoluna, com ícones pequenos opcionais.
wxLC_REPORT	Visualização em *report* simples ou multicoluna com cabeçalho opcional.
wxLC_ICON	Ícones grandes com *labels* opcionais.
wxLC_SMALL_ICON	Ícones pequenos com *labels* opcionais.
wxLC_ALIGN_TOP	Alinha os ícones ao topo (Win32 default, Win32 apenas).
wxLC_ALIGN_LEFT	Ícones são alinhados à esquerda.
wxLC_AUTOARRANGE	Ícones arranjam- se automaticamente.
wxLC_USER_TEXT	A aplicação providencia textos de rótulo sob demanda, exceto para os cabeçalhos.
wxLC_EDIT_LABELS	Permite que os rótulos sejam editáveis.
wxLC_NO_HEADER	Sem cabeçalhos em modo *report*. (Win32 apenas).
wxLC_SINGLE_SEL	Seleção simples.
wxLC_SORT_ASCENDING	Classifica em ordem ascendente.
wxLC_SORT_DESCENDING	Classifica em ordem descendente.
wxLC_HRULES	Desenha separador horizontal entre linhas.
wxLC_VRULES	Desenha separador vertical entre colunas.

Funções utilitárias:

- listCtrlSetImageList :: ListCtrl a ImageList b Int IO (): Permite especificar um ImageList para servir de fonte de imagens para o *widget*. O parâmetro inteiro deve ser uma das seguintes constantes: wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL.
  Obs: O ImageList **não** será destruído quando o ListControl for destruído.

- listCtrlAssignImageList :: ListCtrl a ImageList b Int IO (): Permite especificar um ImageList para servir de fonte de imagens para o *widget*. O parâmetro inteiro deve ser uma das seguintes constantes: wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL.
  Obs: ImageList será destruído quando o ListControl for destruído.

Instâncias e atributos relevantes:

- listEvent
  - listEvent :: Event (ListCtrl a) (EventList IO ()): Define o evento de lista.
- columns
  - columns :: Attr (ListCtrl a) [(String, Align, Int)]: Atributo que define os cabeçalhos das colunas.
- Items
  - Para o ListControl, itens tem o tipo [String]

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.21 – Tree Control

Uma TreeControl é uma representação gráfica de uma árvore, na qual se pode expandir e retrair os nós (muito utilizada em navegadores de sistemas de arquivos como, por exemplo, o Windows Explorer).

Criação:

- treeCtrl :: Window a [Prop (TreeCtrl ())] IO (TreeCtrl ()): Cria uma TreeControl simples, com botões para expandir e retrair nós.
- treeCtrlEx :: Window a Style [Prop (TreeCtrl ())] IO (TreeCtrl ()): Cria uma TreeControl com o estilo especificado.

Styles:

Nome da constante	Descrição
wxTR_EDIT_LABELS	Use este estilo para permitir que o usuário modifique os *labels*.
wxTR_NO_BUTTONS	Us este estilo no caso em que não há interesse em desenhar botões.
wxTR_HAS_BUTTONS	Desenha botões + e - à esquerda dos nós que possuem filhos.
wxTR_TWIST_BUTTONS	Use este estilo para mostrar botões *twist*, no estilo Mac, à esquerda dos itens pai.
wxTR_NO_LINES	Esconde linhas conectoras verticais.
wxTR_FULL_ROW_HIGHLIGHT	Use este estilo para estender a cor de fundo e a seleção sobre toda a linha vertical onde está o item. (Este *flag* é ignorado sobre o Windows a menos que o flag wxTR_NO_LINES seja ligado)

wxTR_LINES_AT_ROOT	Mostra linhas entre nós pai. É aplicável apenas se wxTR_HIDE_ROOT estiver ligado e wxTR_NO_LINES não estiver ligado.
wxTR_HIDE_ROOT	Suprime a exibição do nó raiz, fazendo com que seus nós filhos sejam mostrados como uma seqüência de nós raiz.
wxTR_ROW_LINES	Exibe uma borda de restrição ao redor das linhas.
wxTR_HAS_VARIABLE_ROW_HEIGHT	Faz com que as linhas ajustem sua altura de acordo com o conteúdo. Quando desligado, todas as linhas terão a maior dentre as alturas.O padrão é desligado.
wxTR_SINGLE	Permite que apenas um único item seja selecionado (padrão).
wxTR_MULTIPLE	Permite a seleção de uma faixa de itens.
wxTR_EXTENDED	Permite a seleção disjunta de itens. (Não implementado por completo. Pode falhar em alguns casos).
wxTR_DEFAULT_STYLE	Estilo padrão.

Funções utilitárias:

- treeCtrlAddRoot :: TreeCtrl a String Int Int TreeItemData e IO TreeItem : Adiciona um nó raiz com o string passado como argumento. O argumento de tipo TreeItemData pode ter o valor objectNull. Retorna o TreeItem adicionado. Os dois argumentos inteiros são, respectivamente, o índice de uma imagem (em um ImageList) para ser exibida quando o item não está selecionado, e o índice de uma imagem (em um ImageList) para ser exibido quando o item estiver selecionado. Ambos podem ter o valor -1, se não houver nenhuma imagem.
- treeCtrlAppendItem :: TreeCtrl a TreeItem String Int Int TreeItemData f IO TreeItem: Adiciona, como filho do TreeItem passado como argumento, um nó contendo a string passada como argumento.
- treeCtrlAssignImageList :: TreeCtrl a ImageList b IO (): Associa uma ImageList ao controle. Quando o controle for destruído, a ImageList também será deletada.
- treeCtrlSetImageList :: TreeCtrl a ImageList b IO (): Define a ImageList como fonte de imagens. Quando a TreeControl for deletada, a ImageList não será.
- treeCtrlDelete :: TreeCtrl a TreeItem IO () : Deleta, da TreeControl, o TreeItem passado como argumento.

- treeCtrlDeleteAllItems :: TreeCtrl a TreeControl IO (): Deleta todos os item da TreeControl.

Instâncias e atributos relevantes:

- treeEvent: Define os eventos da TreeControl.
    - treeEvent :: Event (TreeCtrl a) (EventTree -> IO ())

Lista de instâncias: Textual, Literate, Dimensions, Colored, Visible, Child, Able, Tipped, Identity, Styled, Reactive, Paint.

2.22 – Timer:

O Timer é um "controle" não visual cujo propósito é disparar um evento padrão – command – sempre que um determinado intervalo de tempo tiver decorrido.

Criação:

- timer :: Window a [Prop Timer] IO Timer : Cria um Timer.

Instâncias e atributos relevantes:

- interval
    - interval :: Attr Timer Int : Define o intervalo de tempo no qual o evento command é disparado.
- Able
- Commanding

2.23 – Menu

Como vimos anteriormente, podemos adicionar um menu a um Frame especificando uma lista menu por meio do atributo menuBar :: WriteAttr (Frame a) [Menu ()].

Criação:

- menuPane :: [Prop (Menu ())] V IO (Menu ()): Esta função permite criar um *container* para menuItens. Podemos exibir este menu de duas formas, como veremos adiante.
- menuItem :: Menu a [Prop (MenuItem ())] IO (MenuItem ()): Cria item de menu.

- menuSub :: Menu b Menu a [Prop (MenuItem ())] IO (MenuItem ()): Cria um menuItem em o argumento do tipo Menu a contém um submenu, especificado pelo argumento do tipo Menu b.

Obs: O campo de texto de MenuItem pode conter atalhos para os menus, como por exemplo: menuItem menu [text := "&Open\tCtrl+O", help := "Opens an existing document"] o "&" antes do "Open" tornará a primeira letra da string um atalho para este item de menu. O "\tCtrl+O" liga o item de menu ao evento de teclado Ctrl+O.

Funções utilitárias:

- menuPopup :: Menu b Point Window a IO (): Mostra um menu pop-up em um determinado ponto de uma janela.

Instâncias e atributos relevantes:

- menu :: MenuItem a Event (Window w) (IO ()): Evento de menu, que ocorre quando um dado menuItem é clicado.
- menuId :: Id Event (Window w) (IO ()): Reage a um evento de menu baseado em uma identidade.

Lista de instâncias: Textual, Able, Help, Checkable, Identity, Commanding.

Obs: estas instâncias referem-se apenas ao MenuItem.

2.24 – Tool bar

Criação:

- toolBar :: Frame a [Prop (ToolBar ())] IO (ToolBar ()): Cria uma barra de ferramentas vazia, com legendas.
- toolBarEx :: Frame a Bool Bool [Prop (ToolBar ())] IO (ToolBar ()): Cria uma barra de ferramentas vazia. O primeiro argumento informa se devem ser mostradas legendas. O segundo argumento informa se uma linha divisora dever ser exibida acima da barra de ferramentas.
- toolMenu :: ToolBar a MenuItem a String FilePath [Prop ToolBarItem] IO ToolBarItem: Cria um item de barra de ferramentas baseado em um item de menu. Tem como argumentos o item de menu ao qual se deseja ligar uma legenda e uma imagem (.bmp,.gif.,.ico etc) geralmente de tamanho 16x15 pixels.

- toolItem :: ToolBar a String Bool FilePath [Prop ToolBarItem] IO ToolBarItem: Cria um item de barra de ferramentas que não está associado a um item de menu. Tem como argumentos uma legenda, um atributo booleano, que é verdadeiro quando o botão é *checkable*, e uma imagem para ser exibida como ícone.
- toolControl :: ToolBar a Control b IO (): Adiciona um controle à barra de ferramentas (geralmente um comboBox). Para que isto funcione adequadamente o controle deve ser criado com a barra de ferramentas como pai.

Instâncias e atributos relevantes:

- tool :: ToolBarItem Event (Window w) (IO ()): Evento de item de barra de ferramentas. *Obs*: Para itens de barra de ferramenta associados a um menu, não é necessário definir um tratador de evento, basta que o tratador de evento de item de menu esteja definido.

Lista de instâncias: Textual, Able, Help, Tipped, Checkable, Identity, Commanding.

2.25 – Status Bar

Tal como acontece com o menu, uma barra de status pode ser adicionada ao Frame por meio do atributo statusBar :: WriteAttr (Frame a) [StatusField].

Criação:

- statusField :: [Prop StatusField] IO StatusField: Cria um campo de barra de status.

Instâncias e atributos relevantes:

- statusWidth :: Attr StatusField Int: Permite definir o tamanho (comprimento) de um campo da barra de status. Um valor negativo fará com que o campo seja extensível, em relação ao espaço que é deixado pelos outros campos.

Lista de instâncias: Textual.

3 – Diálogos

3.1 – Diálogos predefinidos:

3.1.1 - Mensagens

- errorDialog :: Window a String String IO (): Exibe uma mensagem de erro com um botão ok. Ex: errorDialog parent "error" "fatal error, please re-install Windows"

- warningDialog :: Window a String String IO (): Diálogo de advertência com um botão "Ok". Ex: warningDialog parent "warning" "you need a break"

- infoDialog :: Window a String String IO (): Um diálogo de informação com um botão "Ok". Ex: infoDialog parent "info" "you've got mail"

- confirmDialog :: Window a String String Bool IO Bool: Um diálogo de confirmação sobre alguma tarefa. Possui dois botões "Yes" e "No". Se o argumento booleano for True, o botão "Yes" será a opção padrão. Caso contrário, a opção padrão será o botão "No". Este diálogo irá retornar True quando o botão "Yes" tiver sido pressionado. Ex: yes <- confirmDialog parent "confirm" "are you sure that you want to reformat the hardisk?" True

- proceedDialog :: Window a String String IO Bool: um diálogo de procedência. Exibe os botões "Ok " e "Cancel". Retorna True, quando "Ok" tiver sido pressionado. Ex: ok <- proceedDialog parent "Error" "Do you want to debug this application?"

3.1.2 – Arquivos:

- fileOpenDialog :: Window a Bool Bool String [(String, [String])] FilePath FilePath IO (Maybe FilePath) : Mostra um diálogo modal de seleção de arquivos. Um diálogo modal é aquele em o usuário não pode realizar nenhuma outra operação na aplicação, até que o diálogo seja fechado.

Exemplo:

fileOpenDialog parent rememberCurrentDir allowReadOnly message wildcards directory filename

O argumento rememberCurrentDir dever ser True quando for desejável que o diálogo "lembre" qual o último diretório acessado. O argumento allowReadOnly serve para especificar quando arquivos ReadOnly poderão ser selecionados. O argumento wildcards determina as entradas na lista de seleção de tipos de arquivos. Essa lista é, na verdade, uma lista de pares, onde o primeiro componente do par fornece uma descrição do tipo de arquivo, e o segundo

componente é uma mascara, ou filtro, para o tipo de arquivo. Os dois últimos argumentos são o diretório inicial (“” = diretório corrente) e o nome de arquivo padrão (“” = nenhum).

Exemplo:

```
fileOpenDialog frame True True "Open image" [("Any file",["*.*"]),("Bitmaps",["*.bmp"])] "" ""
```

O resultado será Just FilePath quando o usuário pressionar “OK” e Nothing quando o usuário pressionar “Cancel”

- filesOpenDialog :: Window a Bool Bool String [(String, [String])] FilePath FilePath IO [FilePath]: Diálogo de abrir arquivo que permite ao usuário selecionar mais de um arquivo para ser aberto. Retorna [], quando o usuário clica em “Cancel”.

- fileSaveDialog :: Window a Bool Bool String [(String, [String])] FilePath FilePath IO (Maybe FilePath) : Mostra um diálogo modal “Salvar Arquivo”.

Exemplo: fileSaveDialog parent rememberCurrentDir overwritePrompt message directory filename

O argumento overwritePrompt determina quando o usuário deve ser avisado sobre subscrição de um arquivo. Os demais argumentos são como no diálogo anterior.

- dirOpenDialog :: Window a Bool FilePath FilePath IO (Maybe FilePath): Exibe um diálogo modal “Diretório”.

Exemplo: dirOpenDialog parent allowNewDir message directory

O argumento allowNewDir determina se o usuário poderá criar um novo diretório ou modificar os nomes dos diretórios existentes. O argumento message é apenas para ser exibido no topo da janela.

3.1.3 – Miscelânia:

- fontDialog :: Window a FontStyle IO (Maybe FontStyle): Diálogo para seleção de fonte.

- colorDialog :: Window a Color IO (Maybe Color): Diálogo para seleção de cor.

- passwordDialog :: Window a String String String IO String: Diálogo para entrada de senhas. Retorna “” se cancelado. O último argumento é o texto padrão.

- textDialog :: Window a String String String IO String : Diálogo para entrada de um texto pelo usuário. Retorna "" se cancelado.

- numberDialog :: Window a String String String Int Int Int IO (Maybe Int) : Diálogo para entrada numérica: Retorna Nothing se cancelado.

Exemplo: numberDialog window message prompt caption initialValue minimum maximum

3.2 – Escrevendo os próprios diálogos:

Assim como temos a função frame, para criar uma janela, temos a função diálogo para criar um diálogo.

```
dialog :: Window a -> [Prop (Dialog ())] -> IO (Dialog ())
```

A principal diferença entre um diálogo e um frame, é que o diálogo é mais limitado que o frame, isto é, não permite barras de ferramentas e menus. Além disso, um diálogo pode ser exibido de duas maneiras distintas. A maneira convencional (usando a propriedade visible para exibir/ocultar o menu). A maneira modal, onde o usuário apenas poderá prosseguir na aplicação após ter concluído o diálogo. Para exibir um diálogo de maneira modal, usamos a função showModal:

```
showModal :: Dialog b -> ((Maybe a -> IO ()) -> IO ()) -> IO (Maybe a)
```

A função passada como argumento serve para parar o diálogo. O exemplo a seguir ilustra o uso de um diálogo modal.

```
module DialogSmaples where

import Graphics.UI.WX
import Graphics.UI.WXCore

main = start gui

userDialog :: Window a    IO (Maybe (String,String))
userDialog w = do d ← dialog w [text := "Login de usuário"]
                  p ← panel d []
                  name ← entry p []
                  passwrd ← entry p [style := password]

                  bottom ← panel d []
```

```
                btOk ← button bottom [text := "Ok"]
                btCancel ← button bottom [text := "Cancel"]

                set bottom [layout := row 2 [widget btCancel, widget btOk]]

                set p [layout := column 2 [row 2 [label "Nome:", widget name],
                                           row 2 [label "senha:", widget passwrd]] ]
                set d [layout := margin 5 (column 3 [widget p,widget bottom])]

                showModal d (\stop -> do set btOk [on command :=
                                                    do n ← get name text
                                                       pass ← get passwrd text
                                                            stop (Just (n,pass))]

                                         set btCancel [on command := stop (Nothing)])

 where password = wxTE_PROCESS_ENTER .+. wxTE_PROCESS_TAB .+. wxTE_PASSWORD

gui = do f ← frame [text := "Exemplo sobre diálogos "]
         p ← panel f []
         b ← button p [text := "Login",on command := login f]
         set p [layout := margin 5 (column 2 [label "Clique aqui para logar", widget b ]) ]
         set f [layout := widget p]

 where
 login f = do r ← userDialog f
              case r of
               (Just (name, password))
                       if ((name == "programmer") && (password == "haskell"))
                          then infoDialog f "Login" "Login efetuado"
                          else errorDialog f "Login" "Senha ou usuário incorreto"

              (Nothing)                  return ()
```

4 – O módulo Draw

O modulo Graphics.UI.WX.Draw é automaticamente importado quando se importa o modulo Wx. Esse módulo define as funções básicas para desenhar em objetos especiais chamados *Device Context* ou simplesmente DC. Praticamente todo objeto gráfico está associado a algum DC. Uma maneira relativamente simples de ter acesso ao objeto DC de algum componente gráfico é definir o método paint para este objeto. Lembre- se da assinatura do método paint:

```
paint :: Paint w => Event w (DC ()   Rect    IO ())
```

Ou seja, o tratador deste evento é uma função que toma o DC e uma região retangular da tela e realiza alguma operação.

Draw é, na verdade, uma classe, e o único objeto correspondente de WxWindows que instancia esta classe é o DC. Mas Draw não é a única classe instanciada por este objeto. Outras classes são: Brushed, Literate, e Colored. Veremos para que servem estas classes mais adiante.

4.1 – Atributos de DC oriundos da classe Draw

```
penKind :: Drawn w => Attr w PenKind
```

O PenKind é o tipo de linha, ou borda, que estamos desenhado, ou ainda, se você preferir, o tipo de "ponta da caneta" . Seus construtores são:

- PenTransparent: Sem borda.
- PenSolid: Linha contínua
- PenDash: Linha tracejada.
  - _penDash :: !DashStyle: Onde DashStyle pode ser:
    - DashDot: Pontilhado
    - DashLong: Traçado Longo.
    - DashShort: Traçado Curto
    - DashDotShort: Pontilhado curto.
- PenHatch
  - _penHatch :: !HatchStyle onde HatchStyle pode ser:
    - HatchBDiagonal: Diagonal invertida
    - HatchCrossDiag: Diagonal cruzada
    - HatchFDiagonal: Diagonal
    - HatchCross: Cruz ortogonal.
    - HatchHorizontal: Horizontal.
    - HatchVertical: Vertical

- PenStipple: _penColor é ignorado
  - _penBitmap :: !(Bitmap ()): Usa um bitmap como tipo de ponta.

```
penWidth :: Drawn w => Attr w Int
```
: O tamanho do ponto gráfico.

```
penCap :: Drawn w => Attr w CapStyle
```

Refere-se às extremidades dos pontos:

- CapRound: Extremidade arredondadas
- CapProjecting: Extremidades projetadas
- CapButt: Extremidades grossas.

penJoin :: Drawn w => Attr w JoinStyle

Refere-se as quinas dos pontos:

- JoinRound: Cantos arredondados
- JoinBevel: Cantos esquadrejados
- JoinMiter: Cantos em bloco.

penColor :: Drawn w => Attr w Color : Define a cor da caneta, ou seja, a cor das linhas que serão desenhadas.

pen :: Drawn w => Attr w PenStyle: Por fim uma PenStyle é um aglomeração de todos os atributos vistos anteriormente:

- _penKind :: !PenKind
- _penColor :: !Color
- _penWidth :: !Int
- _penCap :: !CapStyle
- _penJoin :: !JoinStyle

Alguns PenStyles já estão predefinidos:

- penDefault :: PenStyle: Caneta padrão (PenStyle PenSolid black 1 CapRound JoinRound)
- penColored :: Color -> Int -> PenStyle: Caneta com cor e espessura especificadas.
- penTransparent :: PenStyle: Caneta transparente.

4.2 – Atributos de DC oriundos da classe Brushed

A classe Brushed refere-se ao estilo de preenchimento usado em uma figura. Os seguintes atributos definem o estilo de preenchimento.

brushKind :: Brushed w => Attr w BrushKind : BrushTransparent No filling

- BrushSolid: Preenchimento com cor sólida.
- BrushHatch: Preenchimento com padrão.
  - _brushHatch :: !HatchStyle : Os mesmos que vimos anteriormente.
- BrushStipple: Padrão de bitmap. (no win95 apenas bitmaps 8x8 são suportados)
  - _brushBitmap :: !(Bitmap ())

brushColor :: Brushed w => Attr w Color: Uma cor de preenchimento.

brush :: Brushed w => Attr w BrushStyle : Assim como Pen, o Brush é uma aglomeração dos estilos anteriores.

- _brushKind :: !BrushKind
- _brushColor :: !Color

4.3 – funções para desenho de formas básicas:

- circle :: DC a Point Int [Prop (DC a)] IO (): Desenha um círculo com centro e raio especificados. O valor do tipo Point pode ser facilmente construído por meio da função pt :: Int Int Point.

- arc :: DC a Point Int Double Double [Prop (DC a)] IO (): Desenha um arco de circunferência. Devem ser fornecidos como argumentos o centro, o raio, um ponto inicial e um ponto final, os dois últimos relativos a posição 0 do arco trigonométrico. Ângulos são marcados em graus e valores positivos indicam movimento no sentido horário.

- ellipse :: DC a Rect [Prop (DC a)] IO (): Desenha uma elipse restringida por uma região retangular .

- ellipticArc :: DC a Rect Double Double [Prop (DC a)] IO (): Desenha um arco elíptico. O argumento de tipo Rect é o retângulo que restringe a elipse. Os demais argumentos são como na função arc.

- line :: DC a -> Point Point [Prop (DC a)] IO (): Desenha uma linha.

- polyline :: DC a [Point] [Prop (DC a)] IO (): Desenha um poly- linha.

- polygon :: DC a [Point] [Prop (DC a)] IO (): Desenha um polígono, contornado com a regra impar- par. Note que um polígono é automaticamente fechado.

- drawPoint :: DC a Point [Prop (DC a)] IO (): Desenha um ponto.

- drawRect :: DC a Rect [Prop (DC a)] IO (): Desenha um retângulo.

- roundedRect :: DC a Rect Double [Prop (DC a)] IO (): Desenha um retângulo de cantos arredondados. Os cantos são quartos de círculos com um raio dado. Se o raio é positivo, é considerado o valor do raio dos cantos. Se o

valor for negativo, é considerado o valor absoluto de uma proporção da menor dimensão do retângulo.

- drawText :: DC a String -> Point [Prop (DC a)] IO () : Desenha um texto.
- rotatedText :: DC a String Point Double [Prop (DC a)] IO () : Desenha um texto rotacionado, tendo a inclinação como argumento.
- drawBitmap :: DC a Bitmap () Point Bool [Prop (DC a)] IO (): Desenha um bitmap. Tem como argumentos o bitmap, a posição de onde o desenho se inicia, e um argumento que deve ser True, para desenhar o bitmap com máscara de transparência.
- drawImage :: DC a Image b Point [Prop (DC a)] IO (): Desenha uma imagem.

5 – Processamento de Som e imagens

Infelizmente, *WxHaskell* não provê muitos recursos para processamento multimídia. Os recursos existentes para manipulação de som são bem limitados.

5.1 – Recursos para imagens:

É possível utilizar grande variedade de imagens baseadas em rgb (.gif,.jpeg,.bmp,.png etc).

- image :: FilePath Image (): Carrega uma imagem de um arquivo. A única instância do tipo Image é Sized, uma classe que define os atributos de tamanho.
- imageCreateFromFile :: String IO (Image ()): Função alternativa para carregar uma imagen de um arquivo.
- imageCreateFromPixels :: Size -> [Color] IO (Image ()): Cria uma imagem a partir de uma lista de pixels.
- imageGetPixels :: Image a IO [Color]: Retorna a lista de pixels de uma imagem.
- imageCreateFromPixelArray :: Array Point Color IO (Image ()): Cria uma imagem a partir de um array de pixels.

- imageGetPixelArray :: Image a → IO (Array Point Color): Retornas os pixels de uma imagem como um array de pixels.

- bitmap :: FilePath → Bitmap (): Retorna um bitmap lido de um arquivo. Extensões como .ico, .xpm ou png também podem ser carregadas neste tipo.

- bitmapCreateFromFile :: FilePath → IO (Bitmap ()): Carrega um bitmap de um arquivo.

- bitmapFromImage :: Image a → IO (Bitmap ()): "Converte" um valor do tipo Image em um valor do tipo Bitmap.

5.2 – Recursos para Som:

- sound :: FilePath -> Wave (): Carrega um arquivo de som na memória.

- play :: Wave a -> IO (): Reproduz um Wave de forma assíncrona, isto é, envia o sinal para o sistema reproduzir o som e continua imediatamente a execução do programa.

- playLoop :: Wave a → IO (): Reproduz um fragmento de som repetidamente e de forma assíncrona.

- playWait :: Wave a → IO (): Reproduz um arquivo de som de forma síncrona, isto é, envia o sinal para o sistema reproduzir o som, aguarda o termino da execução, e só então continua a aplicação.

6 – Referências

1) Haskell: http://www.haskell.org/
2) WxHAskell: http://wxhaskell.sourceforge.net/
3) WxWindows: http://www.wxwindows.org
4) FFI (Forein Function Interface): http://www.cse.unsw.edu.au/~chak/haskell/ffi/