TclHaskell – user manual

Meurig Sage
(Based on first version by Chris Dornan)

Aug 4, 1999

1 Introduction

The aim of the TclHaskell package is to provide Haskell programmers with the means of building graphical user interfaces through the Tcl/Tk. This user manual explains the basic ideas involved, and so assumes a reasonable knowledge of Haskell. [2] For further information on Tcl/Tk see [1]. Tcl/Tk also includes extensive on-line help. For a full set of TclHaskell demos run the main function in Demo.hs in the demos directory.

2 Getting Started

A simple Tcl script to place a button in a new top-level window, which is removed when the button is pressed or the escape key is pressed over the button, could be written as follows:

```
# a simple Tcl script
toplevel .hello
wm title .hello Hello
button .hello.b -text Hello -command {destroy .hello}
bind .hello.b <Key-Escape> {destroy .hello}
focus .hello.b
pack .hello.b
```

This can be reexpressed as the TclHaskell program:

```
module Main where
import Tcl

main :: IO ()
main = start hello

hello :: GUI ()
hello = do
 top <- window []
 title top title “Hello”]
 let goodbye = destroy top
 but <- button top [text "Hello", command goodbye]
 bind but "<Key-Escape>" goodbye
 focus but
 packAdd but []
```

All the definitions needed to drive TclHaskell are exported from the Tcl module, so this module should be imported into modules that use TclHaskell. Apart from the import declaration, three procedures are declared: main, hello and goodbye. The main procedure is of type IO (), as required by Haskell and it simply invokes the hello procedure, of type IO (), as an argument to the start higher-order TclHaskell procedure. All TclHaskell programs should follow this format: place the TclHaskell program in a definition of type GUI (), as was done with the hello definition above, and use start to convert it to a standard Haskell IO program inside the main procedure.

The start function is of type `GUI () -> IO ().` It executes its argument function, which should be regarded as an initialisation procedure; having executed the initisialisation procedure, it enters the TclHaskell event loop, repeatedly servicing events until the main window is destroyed, at which point it shuts down the TclHaskell run-time system and returns.

So the hello procedure is in fact an initialisation procedure (as was the above Tcl script, of course, as the wish Tcl/Tk interpretter enters its event loop after executing the script).

The calls made in the hello world program fall into three categories: widget creation procedures (window and button), support procedures (title), event binding procedures (bind) and packing procedures (pack) to display a widget on a screen.

The widget creation procedures window and button are Haskell wrappers for their Tk namesakes, and they take their arguments in the same order. TclHaskell automatically generate unique tcl names for the widgets. Buttons are children of windows so they take their window parent as an argument. If you need to get access to this unique name you can do so using `wpath` which takes a widget and returns its String name. If you need to give a TclHaskell widget a specific name you can do so. This can be useful if you want to refer to the TclHaskell widget inside some Tcl code, without passing the name around explicitly. For instance, we could have given the window the name .hello using the creation function window'.

```
hello = do
 top <- window' ".hello" []
 …
```

The configuration options (text and command) consist of functions with names similar to the corresponding Tk options, so the Tk command

```
button .hello.b -text Hello -command --destroy .hello
```

is translated into TclHaskell:

```
button ".hello.b" [text "Hello", command goodbye]
```

The types of the arguments to configuration functions like text and command follow the arguments of the corresponding Tk options. Haskell types are used instead of String most of the time. So, for instance, with one where the Tk option would normally take a script to be executed in response to a prescribed event, the TclHaskell option function will take a Haskell `GUI ()` action to be executed instead. So when the button is pressed while the TclHaskell program is running, the goodbye action is invoked. The widget creation procedures also return Haskell data structures to be used with other TclHaskell procedures such as `title` and `bind`. The `title` procedure is a straight analogue of the `wm title` Tk command, while bind performs the same function as the bind Tk command, except that it invokes GUI procedures instead of Tcl scripts.

The remainder of this paper systematically introduces the functions and types provided by the Tcl module under the following headings: the GUI monad, calling Tcl, widget creation, binding, state and 'odds and ends'.

3 The GUI Monad

```
data GUI a
instance Functor GUI
instance Monad GUI
start :: GUI () -> IO ()
quit :: GUI ()
proc :: IO a -> GUI a
failGUI :: IOError -> GUI a
tryGUI :: GUI a -> GUI (Either IOError a)
catchGUI :: GUI a -> (IOError -> GUI a) -> GUI a
```

The GUI monad is essentially an extension of the I/O monad. As explained in section 2 the start procedure can be used to run GUI programs, terminating when the root window (.) is destroyed. The quit procedure can be called at any time to kill the root widget and so shutdown the TclHaskell session, the Haskell program start quit does nothing, as the initialisation procedure simply kills the automatically created root widget and ending the session. (The Haskell program start (return ()) starts up, creating the root widget, which must be dismissed through the window manager before the program is completes.)

The proc procedure can be used to run I/O procedures from GUI procedures. Note that all GUI procedures, including the initilisation procedure, should complete their task quickly so that the TclHaskell event loop can be established

quickly and remain running more or less continuously. As such I/O commands like getLine that block indefinitely should not be called from GUI procedures.
The GUI monad is an error-handling monad, just like the IO monad, so failGUI, tryGUI and catchGUI perform the same functions for the GUI monad as the ioError, try and catch do for the IO monad.

4 Calling Tcl

If you need to call abritrary Tcl code explicitly you can do so using the `tcl` function.

```
tcl :: [String] -> GUI String
tcl_ :: [String] -> GUI ()
```

This can be helpful if you need to use an obscure tcl-tk feature not currently supported by TclHaskell. However, most of the time it will be uneccessary. The argument to these procedures is a list of Tcl strings, which are joined together with separating spaces (i.e., with unwords), with the resulting Tcl script being passed to the Tcl interpreter for execution. With tcl, the result of the command is returned. The following script illustrates the tcl command, where the Tcl clock command is used first to get the time and then to extract the hours, minutes and seconds components, returning them as integers in a tuple.

```
getTime :: GUI (Int,Int,Int)
getTime =
do secs <- tcl [“clock seconds”]
   h <- tcl [“clock format”,secs,”-format ``”%I``”“]
   m <- tcl [“clock format”,secs,”-format ``”%M``”“]
   s <- tcl [“clock format”,secs,”-format ``”%S``”“]
   return (parseInt h,parseInt m,parseInt s)

(The parseInt procedure is provided for parsing Tcl-generated integers.)
```

5 Widget Configuration

In section 2 it was shown how the button Tk command,

```
button .hello.b -text Hello -command --destroy .hello
```

was translated into the TclHaskell procedure call:

```
button’ “.hello.b” [text “Hello”, command goodbye]
```

The widget creation procedures (button and friends) will be covered by the following section, but first the functions for setting the widgets’ options (text, command and friends) will be described.

The type of button is

```
button’ :: WPath -> [Conf But] -> GUI Button
```

and the Button type is in fact a type synonym:

```
type Button = Widget PClass But
```

Although the button procedure will be used as an exemplar in this section, all the widget creation procedures have a similar type signature and work in essentially the same way where setting their options is concerned. The type signatures of the text and command functions is almost (see below)

```
text :: String -> Conf w
command :: GUI () -> Conf w
```

so the above application of button is type correct as text is applied to a string and command is applied to a GUI procedure, and they both return a value of type Conf w, which is clearly compatible with the type Conf But that is expected.

In fact this is a simplified picture as type classes are used to ensure that only options that are appropriate to the widget being configured are given. The type signatures of the above procedures are in fact

```
text :: Has_text w => String -> Conf w
command :: Has_command w => GUI () -> Conf w
```

and the same pattern is used for all other options functions. Now when the text or command functions are applied to generate a Conf w, the w type must be a member of the Has_text or Has_command classes, respectively.

As -text and -command are valid options to the button Tk command, the instance declarations are established to reflect this.

```
instance Has_text But
instance Has_command But
```

The But type does not belong to the Has_tags class as -tags is not a valid option for the button Tk command, so an attempt to use the tags option function, whose actual type signature is

```
tags :: Has_tags w => [String] -> Conf w
```

in the button procedure's Config list will cause a type error.
As well as setting options at widget creation time, options can be set and interrogated after creation with the cset and cget procedures.

```
cset :: Widget c w -> [Conf w] -> GUI ()
cget :: Widget c w -> (d->Conf w) -> GUI String
```

Both of these procedures use the same mechanism to ensure that appropriate options are used with a given widget. The above button creation procedure call could be done in two phases, creating the widget with default options before configuring, as follows:

```
do but <- button' ".hello.b" []
   cset but [text "Hello", command goodbye]
```

The cget procedure takes one of the above configuration functions and always returns a string, which may need to be parsed if it is a number, for instance. The current text configuration of a but button widget can be extracted as follows

```
do lbl <- cget but text
```

The list of options directly supported in TclHaskell is by no means exhaustive. The entire collection of options that can be supplied to Tk widgets is very large and being added to as Tk evolves, so only the most common options have been supported with the above mechanism. It is possible however to supply any given option directly to a Tk widget creation command by means of the %% escape mechanism:

```
(%%) :: String -> String -> Conf w
```

The first (left) argument of the operator is the name of the option without any leading - and the second argument is the value of the option. The return Conf type is polymorphic in w so the operator can be used with any of the widget creation functions and all checking is suspended. Note also that the right argument is inserted into the widget creation command with no quoting so the Tcl parser must parse is as an argument. If the right hand argument were "red" then this is fine, but if it contains spaces then some form of quoting must be used: "red green" would cause a runtime parse error (N.B. the quotes here are Haskell quotes) while "--red green " or "``"red green``"" would not cause any parse errors. The tcl-string function,

```
tcl–string :: String -> String
```

can be used to quote a string so that it will pass through the Tcl parser as a single command argument. For example, the -wraplength option of the button command is supported in TclHaskell. However, it expects an Int, representing a value in pixels. Tcl-tk allows other measurement units such as centimeters. We can use %% here.

```
  cset but ["wraplength" %% "5c"]
```

In this case, the “5c” string should contain a simple integer with no spaces, but in situations where this may not be the case the `tcl-string` procedure can be used to quote it for safe passage though the Tcl parser:

```
cset but [“wraplength” %% tcl—string “5c”]
```

Use the %% operator with care as it suspends all the usual compile time checking provided by TclHaskell.

In the case that a Tk flag that is not supported by TclHaskell takes a Tcl script as an argument, the %# operator can be used. It takes the name of the flag in its left argument and the GUI procedure to be invoked by the Tcl script in its second argument.

The complete list of supported configuration options are shown below.

```
-- Set the color of the widget when active. All colors are just Strings, so to get blue use
“blue”.
class Has_activebackground w where activebackground   :: String -> Conf w
class Has_activeforeground w where activeforeground   :: String -> Conf w
-- anchor widget to direction
class Has_anchor w where anchor   :: Anchor -> Conf w
data Anchor = N | S | E | W | NE | NW | SE | SW | C
 deriving (Eq,Show)
-- set the aspect ratio
class Has_aspect w where aspect :: Int -> Conf w
-- set background color
class Has_background w where background   :: String -> Conf w
-- display a tcl bitmap
class Has_bitmap w where bitmap   :: String -> Conf w
-- set the width of any border
class Has_borderwidth w where  borderwidth   :: Int -> Conf w
--provide a command to perform
class Has_command w where   command   :: (GUI ()) -> Conf w
-- set the cursor when over the widget
class Has_cursor w where  cursor   :: String -> Conf w
-- set foreground color when disabled
class Has_disabledforeground w where disabledforeground :: String -> Conf w
-- should we export selection to clipboard
class Has_exportselection w where exportSelection :: Bool -> Conf w
-- show this char instead of other (eg in entry widget)
class Has_ent_show w where  ent_show   :: Char -> Conf w
-- set selection mode
class Has_selectmode w where selectmode   :: SelectMode -> Conf w
data SelectMode = SingleMode | BrowseMode | MultipleMode
                | ExtendedMode
   deriving (Show,Eq)
-- set fill color
class Has_fill w where fill   :: String -> Conf w
-- set font
class Has_font w where font   :: String -> Conf w
-- foreground color
class Has_foreground w where foreground   :: String -> Conf w
-- height of widget
class Has_height w where  height   :: Int -> Conf w
-- details for highlighted widget
class Has_highlightbackground w where highlightbackground   :: String -> Conf w
class Has_highlightcolor w where highlightcolor   :: String -> Conf w
class Has_highlightthickness w where  highlightthickness   :: Int -> Conf w
-- is it horizontally oriented eg scrollbar
class Has_hor_orient w where hor_orient   :: Bool -> Conf w
-- display a tcl image, must make this through tcl-tk calls
class Has_image w where image :: String -> Conf w
-- indicator on eg checkbuttons
class Has_indicatoron w where  indicatoron   :: Bool -> Conf w
class Has_justify w where  justify   :: Justify -> Conf w
data Justify = LeftJ | RightJ | CenterJ
 deriving (Show,Eq)
-- is the widget active normal or disabled state eg a button
class Has_active_state w where   active_state   :: ActiveState -> Conf w
data ActiveState = Active | Disabled | Normal
```

```
 deriving (Show,Eq)
-- outline color
class Has_outline w where  outline   :: String -> Conf w
class Has_padx w where  padx   :: Int -> Conf w
class Has_pady w where pady   :: Int -> Conf w
class Has_postcommand w where   postcommand   :: (GUI ()) -> Conf w
class Has_relief w where relief   :: Relief -> Conf w
data Relief = Raised | Sunken | Flat | Ridge | Solid | Groove
 deriving (Show,Eq)
-- scale info
class Has_sca_from w where   sca_from   :: Int -> Conf w
class Has_sca_length w where sca_length   :: Int -> Conf w
class Has_sca_to w where sca_to   :: Int -> Conf w

class Has_scrollregion w where   scrollregion   :: Rect -> Conf w
type Rect = ((Int,Int),(Int,Int))
-- selected item attributes eg selected text
class Has_selectbackground w where  selectbackground   :: String -> Conf w
class Has_selectborderwidth w where selectborderwidth   :: Int -> Conf w
class Has_selectcolor w where  selectcolor   :: String -> Conf w
class Has_selectforeground w where selectforeground   :: String -> Conf w
-- when resizing do so in grid units eg text character sizes
class Has_setgrid w where  setgrid   :: Bool -> Conf w
class Has_sliderlength w where  sliderlength   :: Int -> Conf w
class Has_takefocus w where  takefocus   :: Bool -> Conf w
-- tags to refer to widgets by eg canvas items
class Has_tags w where  tags   :: [String] -> Conf w
class Has_text w where  text   :: String -> Conf w
-- interval of ticks on a scale widget
class Has_tickinterval w where  tickinterval   :: Int -> Conf w
-- color in trough (sunken) area eg on scrollbar
class Has_troughcolor w where  troughcolor   :: String -> Conf w
-- which character to underline eg in menu
class Has_underline w where  underline   :: Int -> Conf w
-- use which menu
class Has_use_menu w where  use_menu   :: Menu -> Conf w
-- a specific label for menu items
class Has_wgt_label w where  wgt_label   :: String -> Conf w
-- width of widget
class Has_width w where width   :: Int -> Conf w
should we wrap text around
class Has_wrap w where  wrap   :: Wrap -> Conf w
data Wrap = NoWrap | CharWrap | WordWrap
 deriving (Show,Eq)
class Has_wraplength w where  wraplength   :: Int -> Conf w
```

6 Binding

```
type TkEvent = String
type Remover = GUI ()
bind :: Widget c w -> TkEvent -> GUI () -> GUI Remover
bindxy :: Widget c w -> TkEvent -> ((Int,Int)->GUI ()) -> GUI Remover
bindXY :: Widget c w -> TkEvent -> ((Int,Int)->GUI ()) -> GUI Remover
bindArgs :: Widget c w -> (Bool,TkEvent,String) -> ([String]->GUI ()) -> GUI Remover
```

All TclHaskell widgets, except menu-class widgets (i.e., widgets that are used to make up menu entries) which cannot be programmed to react to events in Tk, and Radio widgets, which are not proper Tk widgets anyway, can be programmed to react to events with the bind procedure. (Note that any attempt to use bind with a menu-class widget will be accepted by the type checker but will result in a run-time error; any attempt use bind with a Radio widget will be trapped by the type checker.) The format of the bind procedure is the same as that for the Tk bind command, with GUI procedures being used instead of Tcl scripts. For example, the program,

```
main :: IO ()
main = start (do w <- rootWin; bind w "<Return>" quit;return ())
```

will start up Tk with the default root window and quit when the return key is pressed in the window. (The rootWin function returns the root window in tcl.)

The remove action returned by the bind action removes the binding and deletes the Haskell callback.

The bindArgs procedure provides access to the % mechanism for receiving event parameters. Tupled with its event specifier is a string containing each of the letters of the % options that are required by the command (the boolean components will be explained below); its event handler GUI procedure that takes a list of strings as its argument, the % event parameters in the same order that they were requested in the first argument. For example, the following program will print out the local x and y coordinates of the mouse location when its left button is pressed in the root window.

```
main :: IO ()
main = start (do w <- rootWin; bindArgs w (False,False,"<1>","xy") click)

click :: [String] -> GUI ()
click l = proc (print l)
```

The first boolean argument will normally be false, indicating that the event should be processed as normal. When the flag is True, after the callback has been invoked to service the event, no further event processing will take place. A single external event can give rise to many event-handlers being invoked according to a fixed priority but specifying True with bindArgs ensures that no futher event handlers will be invoked. (See the tcl-tk help pages or see x18.6 `Conflict Resolution' of Tcl and the Tk Toolkit [1] and the `Event Sequences' section of the Chapter `Binding Commands to events' in Practical Programming in Tcl and Tk [3].)

For example, in the following program an event-handler is registered for the left button event for both the root window widget and the .f frame packed into it. Normally, both events would be triggered when the left mouse button is pressed in the window and the frame, with the frame event-handler taking priority, but the frame event handler was bound with a bindArgs call in which the boolean flag is True so, after each mouse click, it is invoked first but it prevents the event-handler registered for the root window from being triggered.

```
main :: IO ()
main = start setup

setup :: GUI ()
setup = do w <- rootWin
           frm <- frame w [width 100,height 100]
           packAdd frm []
        bindArgs frm (True,False,"<1>","xy") click
           bindArgs w "<1>" click’

click :: [String] -> GUI ()
click l = proc(print l)

click' :: GUI ()
click' = proc (putStr “won't happen``n”)
```

The second boolean argument is also normally false. This causes this specific callback to override any other callbacks already bound to the widget. When set to True it causes the callback to be added to other bindings for the widget.

The bindxy and bindXY procedures are convenience procedures for programming event procedures that need access to the local coordinates and global coordinates, respectively, of the mouse position.

7 Widget Creation

```
data Widget a b
instance Eq Widget

data WClass -- abstract
data PClass -- abstract
data MClass -- abstract
data CClass -- abstract
data TClass -- abstract
data EClass -- abstract

type WWidget w = Widget WClass w
type PWidget w = Widget PClass w
```

```
type MWidget w = Widget MClass w
type CWidget w = Widget CClass w
type TWidget w = Widget TClass w
type EWidget w = Widget TClass w

type WPath = String
wpath :: Widget c w -> WPath

parentWPath :: Widget a b -> WPath
tcl_append :: WPath -> String -> WPath
tcl_newWgtName :: GUI WPath
mkChildOf :: Widget c w -> GUI WPath
mkSibling :: Widget c w -> GUI WPath
```

Each of the widget creation functions takes a standard format, which is quite similar to its coresponding widget creation command. For each widget there are two creation functions one which takes an explicit name for a widget, and one which generates a new name using tcl_newWgtName. In the case of the window widgets, the window and window' functions are used:

```
window :: [Conf Top] -> GUI Window
window' :: WPath -> [Conf Top] -> GUI Window
```

The Conf list is a list of configuration options, as explained in section 5. With the second style the first argument is the Tk path for the widget. The return type of `window` is a `Window` value, a TclHaskell structure that can be used to configure and bind the widget (see section 5 and section 6). The `wpath` procedure can be applied to the Window value (or any widget) to find out the path that Tk uses to identify the widget. The `tcl_append` function can be used to join two Tk paths together. Most of the time it merely concatenates the path, e.g.,

```
tcl_append ".kanga" ".roo" = ".kanga.roo"
```

but it it does behave differently for ::

```
tcl_append "." "kanga.roo" = ".kanga.roo"
```

We can make a new unique name that is a child of another widget using mkChildOf, and a sibling using mkSibling. These make use of tcl_append, tcl_newWgtName and parentWPath.

The type Window is in fact an instance of the Widget c w type, where the c parameter is instantiated with a type representing the class that the widget belongs to, and the w parameter is instantiated with a type representing the particular widget:

```
data Win -- abstract
type Window = WWidget Top

type WTag = String
wtag :: Widget a b -> String
```

There are six widget classes. Every widget has a tk path. It also has a unique WTag. For some classes the WPath and WTag will be the same, for others they will not. The differences are discussed below. The instance of Eq on widgets uses the WTag for each widget.

There are six widget classes.

1. WClass Window widgets that are top level objects. This class includes Window and Menu as menus can be popped up and controlled by the window manager. Every widget in this class has a unique Tk path name. WPath and WTag are therefore the same.
2. PClass Most of the widgets, including Button, belong to this class. These can all be displayed in Windos. Every widget in this class also has a unique Tk path name. WPath and WTag are therefore the same.
3. MClass Items that are added to menus are put in this class. The wpath function returns the name of the menu widget to which the entry belongs when applied to an MClass widget. The wtag produces a unique name composed of its WPath and a unique item Identifier. Attempting to bind a an event to a member of this class will cause a runtime error.

4. CClass Items that are added to canvases belong to this class. The wpath function returns the name of the canvas widget to which the item belongs when applied to an CClass widget. The wtag returns a unique name for the item.
5. TClass This class only contains the tags that are configured in Edit widgets (i.e., Tk text widgets). The wpath function returns the name of the edit widget to which the entry belongs when applied to an TClass widget. The wtag returns a unique name formed from the TagId and edit widget wpath.
6. EClass This class covers embedded widgets inside Tk edit widgets. For instance, you can put a button inside an edit widget. The wpath returns the edit widget's wpath. The wtag returns the wpath of the embedded widget.

There are several operations that we can apply to all widgets. These include binding event (except to menu widgets) and setting configuration options. We can also destroy a widget, and add an action to be run when the widget is destroyed. The destroy action cleans up some internal data structures, runs the widget finalisers and then deletes the widget.

```
destroy :: Widget a b -> GUI ()
addFinaliserW :: Widget a b -> GUI () -> GUI ()
```

7.1.1 Window

```
type Window = WWidget Top
data Win -- abstract
data Geometry = WinSz (Int,Int) --- -- size
              | WinPn (Int,Int) --- -- position
              | WinSzPn (Int,Int) (Int,Int) -- size position
window' :: WPath -> [Conf Win] -> GUI Window
window :: [Conf Win] -> GUI Window
rootWin :: GUI Window
genWindow :: [Conf Win] -> GUI Window
destroy :: Window -> GUI ()
title :: Window -> String -> GUI ()
geometry :: Window -> Geometry -> GUI ()
showWindow :: Window -> GUI ()
hideWindow :: Window -> GUI ()
Options: background, borderwidth, cursor, height,
highlightbackground, highlightcolor, highlightthickness, relief,
takefocus, width.
```

The root window is accessed via the rootWin function.

The title utility procedure sets the title of a window (using wm title) and geometry sets its geometry.

7.2 Menu

7.2.1 Making a menu

```
type Menu = WWidget Men
data Men -- abstract
menu' :: WPath -> [Conf Men] -> GUI Menu
menu :: Has_use_menu w => Widget a w -> [Conf Men] -> GUI Menu
menuSize :: Menu -> GUI Int
popup :: Menu -> (Int,Int) -> GUI ()
tearoff :: Bool -> Conf Men
Options: background, borderwidth, cursor, postcommand, relief.
```

The menu procedure generates a Tk menu with can be posted explicitly with the popup procedureThe tearoff configuration option defines whether a menu can be torn off and kept as a separate window. . A menu must be a child of any widget it is attached to. This is important if we want to add a menu to a menubutton or window. We can arrange for this to be True by using the `menu` creation function which also takes the parent widget as an argument. The parent widget must be able to use that menu. Once created the menu can then be attached to its parent widget by applying the use_menu configuration option to the parent. Items are added to the menu with the mbutton, mradiobutton, mcheckbutton, cascade and separator procedures, which issue the appropriate add commands to the menu and return TclHaskell menu-class widgets. Use menuSize to find the number of elements in the menu.

7.2.2 Menu items

We can create menu items using the following set of functions. There are two types of creation function. Functions that add items at the end of the menu, and functions that insert the menu item just before the item at a particular index, starting at 0. Do not use bind to bind an event handler to this widget as this operation is not supported by Tk. The wpath function returns the name of the menu widget containing the button. Use cset and cget to reconfigure and enquire the options of the widget.

7.2.3 Mbutton

```
type MButton = MWidget MBut
data MBut -- abstract
mbutton :: Menu -> [Conf MBut] -> GUI MButton
mbutton' :: Menu -> [Conf MBut] -> GUI MButton
Options: activebackground, activeforeground, background, bitmap,
command, font, foreground, active—state, underline, wgt—label.
```

A basic menu button that performs an action when clicked.

7.2.4 MRadiobutton

```
type MRadiobutton = MWidget MRB
data MRB -- abstract
mradiobutton :: Menu -> [Conf MRB] -> GUI MRadiobutton
mradiobutton :: Menu -> [Conf MRB] -> Int -> GUI MRadiobutton
Options: activebackground, activeforeground, background, bitmap,
command, font, foreground, indicatoron, active—state, selectcolor,
underline, wgt—label.
```

This procedure issues the add radiobutton command to the menu passed. Use the mradio procedure to combine such a collection of radio buttons into a radio group (see section 7.7). Only one of a radio button group can be selected at a time.

7.2.5 MCheckbutton

```
type MCheckbutton = MWidget MChe
data MChe -- abstract
mcheckbutton :: Menu -> [Conf MChe] -> GUI Mcheckbutton
mcheckbutton :: Menu -> [Conf MChe] -> Int -> GUI MCheckbutton
getMCheck :: MCheckbutton -> GUI Bool
setMCheck :: MCheckbutton -> Bool -> GUI ()
varMCheck :: MCheckbutton -> String
Options: activebackground, activeforeground, background, bitmap,
command, font, foreground, indicatoron, active—state, selectcolor,
underline, wgt—label.
```

This procedure issues the add checkbutton command to the menu. Us the getMCheck and setMCheck procedures to set and enquire the check button's status, and the varMCheck function to find out which variable is being used by the check button.

7.2.6 Cascade

```
type Cascade = MWidget CB
data CB -- abstract
cascade :: Menu -> Menu -> [Conf CB] -> GUI Cascade
cascade' :: Menu -> Menu -> [Conf CB] -> Int -> GUI Cascade
Options: activebackground, activeforeground, background, bitmap,
font, foreground, active—state, underline, use—menu, wgt—label.
```

This procedure issues an add cascade command to the menu provided by its first argument. The menu in the second argument is embedded into the first menu and popped up when the cascade button is pressed. This new menu should be a sibling of the menu the cascade added to.

7.2.7 Separator

```
type Separator = MWidget Sep
data Sep -- abstract
separator :: Menu -> GUI Separator
separator' :: Menu -> Int -> GUI Separator
Options: (none).
```

This procedure issues the add separator to a menu. The separator has no configuration options and is just an object to make a gap in the menu.

7.3 PWidgets

PWidgets are packable widgets that can be packed into a window. We can pack a widget into a window using pack or grid layout.

PWidgets can be raised or lowered in stacking order using raise and lower. The raise action raises above a given object if Just or to the top if Nothing; the lower action lowers below a given object if Just to the bottom if Nothing.

```
raise :: PWidget w -> Maybe WPath -> GUI ()
lower :: PWidget w -> Maybe WPath -> GUI ()
```

7.3.1 Frames

To make sub panels in which to place widgets use Frames.

```
type Frame = WWidget Fra
data Fra -- abstract
frame' :: WPath -> [Conf Fra] -> GUI Frame
frame :: Window -> [Conf Fra] -> GUI Frame
Options: height, borderwidth, cursor, relief, width,
highlightbackground, highlightcolor, highlightthickness,
takefocus, background.
```

The use of frames is important here. If we need to place a two horizontal groups of widgets above each other, we place them in two frames (horizontally within each frame) and then place the two frames above each other

7.3.2 Packing Widgets

To place a widget into a window using pack layout use packAdd. To delete it use packForget.

```
packAdd :: PWidget w -> [PackInfo] -> GUI ()
packForget :: PWidget w -> GUI ()
```

The layout is controlled using the PackInfo options. To place widgets horizontally use packH, and vertically use packV.

```
packH,packV :: PackInfo
```

To make a widget fill horizontal space in its parcel use fillX, use fillY for vertical space and use fillXY for both.

```
fillX,fillY,fillXY :: PackInfo
```

To make a widget expand to fill extra space in its master use expand.

```
expand :: Bool -> PackInfo
```

To pad a widget with extra space in x or y dimensions use packPadX and packPadY.

```
packPadX,packPadY :: Int -> PackInfo
```

To make a widget anchor to a particular corner use packAnchor.

```
packAnchor :: Anchor -> PackInfo
```

Note that Anchor was defined when discussing configuration widgets in section 5.

```
data Anchor = N | S | E | W | NE | NW | SE | SW | C
 deriving (Eq,Show)
```

To place a widget at a particular location in the packing order use packPos. If not provided the widget places itself at the end of the packing order.

```
packPos :: PlacePos WPath -> PackInfo

data PlacePos a = PlaceTop
                | PlaceBottom
                | PlaceBefore ! a
                | PlaceAfter ! a
 deriving (Show,Eq)

instance Functor PlacePos
```

By default widgets are placed in their parent. To place a widget inside another frame or window use inFrame, inWindow. Note that a widget can only be placed inside an object that is either its parent or a descendant of one of its ancestors. In general this is useful for creating widgets as children of a particular window and placing them in a frame of that window.

```
inFrame :: Frame -> PackInfo
inWindow :: Frame -> PackInfo
```

7.3.3 Using Grid Layout

To place widgets in a grid where each grid element has a particular size use grid layout. To add and remove use gridAdd and gridForget.

```
gridAdd :: PWidget w -> Coord -> [GridInfo] -> GUI ()
gridForget :: PWidget w -> GUI ()
type Coord = (Int,Int)
```

The coordinate says which column and which row to place the widget in (column first, row second).
The Grid Info options control layout more explictly. To make a widget take up more than one column use widthX. To make a widget take up more than one row use heightY.

```
widthX,heightY :: Int -> GridInfo
```

To pad a widget externally, use gpadX and gpadY. For internal padding use gpadIX, gpadIY.

```
gpadX,gpadY,gpadIX,gpadIY :: Int -> GridInfo
```

To anchor a widget to a particular corner of its cell use gridAnchor.

```
gAnchor :: Anchor -> GridInfo
```

To make a widget fill its cell in X or in Y use gfillX,gfillY,gfillXY.

```
gfillX,gfillY,gfillXY :: GridInfo
```

To place a widget in the grid of an object other than its parent use ginFrame and ginWindow. The standard packing rules (see pack layout) apply here.

```
ginFrame :: Frame -> GridInfo
ginWindow :: Window -> GridInfo
```

7.4 Label

```
type Label = PWidget Lab
data Lab -- abstract
label' :: WPath -> [Conf Lab] -> GUI Label
label :: Window -> [Conf Lab] -> GUI Label
Options: anchor, background, bitmap, borderwidth, cursor, font,
foreground, height, highlightbackground, highlightcolor,
highlightthickness, justify, padx, pady, relief, takefocus, text,
underline, width.
```

Make a label to display text or bitmaps. The `label` function creates a `Label` whose parent is a given window.

7.5 Button

```
type Button = PWidget But
data But -- abstract
button' :: WPath -> [Conf But] -> GUI Button
```

```
button :: Window -> [Conf But] -> GUI Button
Options: activebackground, activeforeground, anchor, background,
bitmap, borderwidth, command, cursor, font, foreground, height,
highlightbackground, highlightcolor, highlightthickness, justify,
active–state, padx, pady, relief, takefocus, text, underline, width.
```

Make a button to perform an action on a command. The `button` function creates a `Button` whose parent is a given window.

7.6 Radiobutton

```
type Radiobutton = PWidget RB
data RB -- abstract
radiobutton' :: WPath -> [Conf RB] -> GUI Radiobutton
radiobutton :: Window -> [Conf RB] -> GUI Radiobutton
Options: activebackground, activeforeground, anchor, background,
bitmap, borderwidth, command, cursor, font, foreground, height,
highlightbackground, highlightcolor, highlightthickness,
indicatoron, justify, active–state, padx, pady, relief, selectcolor,
takefocus, text, underline, width.
```

This option is combined with the radio call to make up groups of radio buttons – see next section. The `radiobutton` function creates a `RadioButton` whose parent is a given window.

7.7 Radio

```
data Radio -- abstract
radio :: [Radiobutton] -> GUI Radio
mradio :: [MRadiobutton] -> GUI Radio
setRadio :: Radio -> Int -> GUI ()
getRadio :: Radio -> GUI Int
varRadio :: Radio -> String
getRadio' :: Radio -> GUI WTag
setRadio' :: Radio -> WTag -> GUI ()
appendMRadio :: Radio -> MRadiobutton -> GUI ()
removeMRadio :: Radio -> MRadiobutton -> GUI ()
appendRadio :: Radio -> Radiobutton -> GUI ()
removeRadio :: Radio -> Radiobutton -> GUI ()
Options: (not applicable).
```

The radio function creates a new Tcl variable and configures all the radio buttons passed to it to use that variable, so combining them into a single group of radio buttons. The mradio procedure does the same for a group of menu radio buttons. setRadio can be used to set a particular button (with 0 signifying the first radio button in the list passed to radio, 1, the second radio button, and so on); the getRadio procedure can be used to find out which button is pressed and varRadio can be used to find the name of the variable being used by the group. The following program will generate a 4 radio buttons, labelled 1-4, pack them horizontally in the root widget and select the first button.

```
radio4 :: GUI ()
radio4 =
do
 w <- window []
 let rb n = do radiobutton w [text (show n)];packAdd w [packH]
 bs <- mapM rb [1..4]
 r <- radio bs
 setRadio r 0
```

Radio buttons can also be added and deleted dynamically from a Radio group, using appendRadio and removeRadio. When doing this it would be very awkward to keep track of which item really was the first or last in the list. The WTag of the radiobutton can instead be used.

7.8 Checkbutton

```
type Checkbutton = PWidget Che
data Che -- abstract
checkbutton' :: WPath -> [Conf Che] -> GUI Checkbutton
checkbutton :: Window -> [Conf Che] -> GUI Checkbutton
getCheck :: Checkbutton -> GUI Bool
setCheck :: Checkbutton -> Bool -> GUI ()
```

```
varCheck :: Checkbutton -> String
Options: activebackground, activeforeground, anchor, background,
bitmap, borderwidth, command, cursor, font, foreground, height,
highlightbackground, highlightcolor, highlightthickness,
indicatoron, justify, active–state, padx, pady, relief, selectcolor,
takefocus, text, underline, width.
```

The checkbutton procedure creates a new Tcl variable and a check button that uses the variable. Us the getCheck and setCheck procedures to set and enquire about the check button's status, and the varCheck function to find out which variable is being used by the check button. The `checkbutton` function creates a `CheckButton` whose parent is a given window.

7.9 Menubutton

```
type Menubutton = PWidget MB
data MB -- abstract
menubutton' :: WPath -> Maybe WPath -> [Conf MB] -> GUI Menubutton
menubutton :: Window -> [Conf MB] -> GUI Menubutton
Options: activebackground, activeforeground, anchor, background,
bitmap, borderwidth, cursor, font, foreground, height,
highlightbackground, highlightcolor, highlightthickness, justify,
active–state, padx, pady, relief, takefocus, text, underline, use–menu,
width.
```

Make a menubutton. The `menubutton'` function creates a menu button and if the second argument is Just m then links the button to menu m. The `menubutton` function creates a menu button is a given window.

7.10 Canvas

```
type Canvas = PWidget Can
data Can -- abstract
type CCoord = String
type Coord = (Int,Int)
instance ScrollableX Can
instance ScrollableY Can
instance Scan Can
canvas' :: WPath -> [Conf Can] -> GUI Canvas
canvas :: Window -> [Conf Can] -> GUI Canvas
citem_canvas :: CWidget a -> GUI Canvas
citem_number :: CWidget a -> Int
moveObject :: CWidget w -> Coord -> GUI ()
removeObject :: CWidget w -> Maybe WTag -> GUI ()
lowerObject :: CWidget w -> Maybe WTag -> GUI ()
raiseObject :: CWidget w -> GUI ()
bboxObjects :: [CWidget a] -> GUI (Int,Int,Int,Int)
getCoords :: CWidget w -> GUI [(Int,Int)]
setCoords :: CWidget w -> [Coord] -> GUI ()
Options: background, borderwidth, cursor, height,
highlightbackground, highlightcolor, highlightthickness, relief,
scrollregion, takefocus, width.
```

A Canvas is an area that may contain graphics canvas items such as lines and ovals as well as other Packable Widgets. To get the parent canvas from a canvas item use citem_canvas. Every canvas item has a unique integer, citem_number. Canvas items can be moved using moveObject. This moves by a particular amount in X and Y directions. To get the bounding box for a canvas use bboxObjects. To get the coordinates of an object use getCoords, to set them use setCoords. They can be raised and lowered in stacking order with raiseObject and lowerObject. (The maybe argument is the tag of another canvas item). Note: these last two commands have no effect on window items in canvases. Window items always obscure other item types, and the stacking order of window items is determined by the raise and lower commands, not the raise and lower widget commands for canvases.

See section 7.12 for the scrollable class. The scan class is discussed in section 7.13.

The following set of items can be displayed in canvases. An object can be created with integer coordinates or using CCoords that are strings. This allows coordinates to be specified in any of the other tcl forms such as centimeters. Canvas items remain visible till destroyed.

7.10.1 COval

```
type COval = CWidget COva
data COva -- abstract
coval' :: Canvas -> CCoord -> CCoord -> [Conf COva] -> GUI COval
coval :: Canvas -> CCoord -> Coord -> [Conf COva] -> GUI COval
Options: fill, outline, tags, width.
```

Creates an oval given an upper left and lower right corner.

7.10.2 CLine

```
type CLine = CWidget CLin
data CLin -- abstract
cline' :: Canvas -> [CCoord] -> [Conf CLin] -> GUI CLine
cline :: Canvas -> [Coord] -> [Conf CLin] -> GUI CLine
Options: fill, tags, width.
```

Create a polyline.

7.10.3 CRectangle

```
type CRectangle = CWidget CRec
data CRec -- abstract
crectangle' :: Canvas -> CCoord -> CCoord -> [Conf CRec] -> GUI CRectangle
crectangle :: Canvas -> Coord -> Coord -> [Conf CRec] -> GUI CRectangle
Options: fill, outline, tags, width.
```

Create a rectangle.

7.10.4 CArc

```
type CArc = CWidget CAr
data CAr -- abstract
carc' :: Canvas -> CCoord -> CCoord -> [Conf CAr] -> GUI CArc
carc :: Canvas -> Coord -> Coord -> [Conf CAr] -> GUI CArc
Options: fill, outline, tags, width.
```

Create an arc.

7.10.5 CPoly

```
type CPoly = CWidget CPol
data CPol -- abstract
cpoly' :: Canvas -> [CCoord] -> [Conf CPol] -> GUI CPoly
cpoly :: Canvas -> [Coord] -> [Conf CPol] -> GUI CPoly
Options: fill, outline, tags, width.
```

Create a polygon.

7.10.6 CText

```
type CText = CWidget CTex
data CTex -- abstract
ctext' :: Canvas -> CCoord -> [Conf CTex] -> GUI CText
ctext :: Canvas -> Coord -> [Conf CTex] -> GUI CText
Options: anchor, fill, font, justify, tags, text, width.
```

Create a text item.

7.10.7 CBitmap

```
type CBitmap = CWidget CBit
data CBit -- abstract
cbitmap' :: Canvas -> CCoord -> [Conf CBit] -> GUI CBitmap
cbitmap :: Canvas -> Coord -> [Conf CBit] -> GUI CBitmap
Options: selectbackground, selectforeground, selectborderwidth,
foreground, justify, anchor, bitmap, tags.
```

Create an item displaying a bitmap.

7.10.8 CImage

```
type CImage = CWidget CIm
data CIm -- abstract
cimage' :: Canvas -> CCoord -> [Conf CBit] -> GUI CBitmap
cimage :: Canvas -> Coord -> [Conf CBit] -> GUI CBitmap
Options: anchor, bitmap, tags.
```

Create an item displaying a bitmap.

7.10.9 CWindow

```
type CWindow = CWidget CWin
data CWin -- abstract
cwindow' :: Canvas -> CCoord -> PWidget w -> [Conf CWin] -> GUI CWindow
cwindow :: Canvas -> Coord -> PWidget w -> [Conf CWin] -> GUI CWindow
Options: anchor, justify, tags.
```

Create a canvas item contain a packable widget.

7.11 Scrollbar

```
type Scrollbar = PWidget Scr
data Scr -- abstract
scrollbar :: WPath -> [Conf Scr] -> GUI Scrollbar
vscroll' :: ScrollableY w => WPath -> PWidget w -> [Conf Scr] -> GUI Scrollbar
hscroll' :: ScrollableX w => WPath -> PWidget w -> [Conf Scr] -> GUI Scrollbar
vscroll :: ScrollableY w => PWidget w -> [Conf Scr] -> GUI Scrollbar
hscroll :: ScrollableX w => PWidget w -> [Conf Scr] -> GUI Scrollbar
Options: width, highlightbackground, highlightcolor, highlightthickness, takefocus,
borderwidth, cursor, relief, background, hor—orient.
```

Makes a scrollbar, horizontal or vertical. The scrollbar will be set up so that it scrolls the given widget. If the vscroll/hscroll version is used then it will be a sibling of the created widget. See next section for the scrollable class.

7.12 Scrollable class

Scrollable widgets are members of the Scrollable class. They can be horizontally scrollable (ScrollableX) or vertically scrollable (ScrollableY).

```
class ScrollableX w
class ScrollableY w

xview :: ScrollableX b => Widget a b -> GUI (Double,Double)
xMoveTo :: ScrollableX b => Widget a b -> Double -> GUI ()
xScroll :: ScrollableX b => Widget a b -> ScrollUnit -> GUI ()

yview :: ScrollableY b => Widget a b -> GUI (Double,Double)
yMoveTo :: ScrollableY b => Widget a b -> Double -> GUI ()
yScroll :: ScrollableY b => Widget a b -> ScrollUnit -> GUI ()
data ScrollUnit = ScrollPages ! Int | ScrollUnits ! Int
```

The xview or yview function returns a pair of values. Each element is a real fraction between 0 and 1; together they describe the horizontal span that is visible in the window. For example, if the first element is .2 and the second element is .6, 20% of the canvas's area (as defined by the -scrollregion option) is off-screen to the left, the middle 40% is visible in the window, and 40% of the canvas is off-screen to the right.

Thew xMoveTo and yMoveTo functions adjust the view in the window so that fraction of the total width of the canvas is off-screen to the left. Fraction must be a fraction between 0 and 1.

The xScroll (yScroll) function shifts the view in the window left or right (up or down) according to the Scroll unit value. If using page units then the view is adjusted in fractions of the widget's width. If using standard units then it is moved in pixels.

7.13 The Scan class

Widgets in the Scan class can be scanned across.

```
class Scan w
scanMark :: Scan w => Widget a w -> Int -> Int -> GUI ()
scanDrag :: Scan w => Widget a w -> Int -> Int -> GUI ()
```

The function scanMark w x y records x and y and the the widgets current view; used in conjunction with later scanDrag commands. Typically this command is associated with a mouse button press in the widget and x and y are the coordinates of the mouse.

The function scanDrag w x y computes the difference between its x and y arguments (which are typically mouse coordinates) and the x and y arguments to the last scan mark command for the widget. It then adjusts the view by 10 times the difference in coordinates. This command is typically associated with mouse motion events in the widget, to produce the effect of dragging the widget at high speed through its window.

7.14 Entry

```
type Entry = PWidget Ent
data Ent -- abstract
instance ScrollableX Ent
instance Scan Ent
entry' :: WPath -> [Conf Ent] -> GUI Entry
entry :: Window -> [Conf Ent] -> GUI Entry
getEntry :: Entry -> GUI String
setEntry :: Entry -> String -> GUI ()
insertEntry :: Entry -> EIndex -> String -> GUI ()
deleteEntry :: Entry -> EIndex -> EIndex -> GUI ()
setICursor :: Entry -> EIndex -> GUI ()
clearEntrySelection :: Entry -> GUI ()
setEntrySelectionAnchor :: Entry -> EIndex -> GUI ()
isEntrySelected :: Entry -> GUI Bool
setEntrySelection :: Entry -> EIndex -> EIndex -> GUI ()
adjustEntrySelection :: Entry -> EIndex -> GUI ()
setToEntrySelection :: Entry -> EIndex -> GUI ()

data EIndex = EIndex Int | EIndexSelStart | EIndexSelEnd | EIndexAnchor
            | EIndexEnd | EIndexAt ! Int ! Int | EFree String

Options: background, borderwidth, cursor, ent_show, font,
foreground, highlightbackground, highlightcolor,
highlightthickness, justify, active—state, relief, takefocus,
textvariable, width.
```

The entry widget is a one line editable text area. The function `getEntry` returns the whole entry String. The function `setEntry` sets the entry with a String. The function `insertEntry` inserts a String at a given point. To delete values from an entry between two indices use `deletEntry`. To set the insertion cursor location use `setICursor`.

The indices are defined as a character location (EIndex), the start and end of the current selection (EIndexSelStart and EindexSelEnd), the end of the entry (EIndexEnd), a particular location in pixels within the entry widget (EindexAt). To give arbitrary tcl code to use as an index use EFree.

The function `adjustEntrySelection` locates the end of the selection nearest to the character given by index, and adjusts that end of the selection to be at index (i.e including but not going beyond index). The other end of the selection is made the anchor point for future select to commands. If the selection isn't currently in the entry, then a new selection is created to include the characters between index and the most recent selection anchor point, inclusive.

The function `clearEntrySelection` clears the selection if it is currently in this widget. If the selection isn't in this widget then the command has no effect. Returns an empty string.

The function `setEntrySelectionAnchor` sets the selection anchor point to just before the character given by index. Doesn't change the selection.

The function `isEntrySelected` is used to find if there is a selection.

The function setEntrySelection sets the selection to include the characters starting with the one indexed by start and ending with the one just before end. If end refers to the same character as start or an earlier one, then the entry's selection is cleared.

The function `setToEntrySelection w to index` behaves as follows. If index is before the anchor point, set the selection to the characters from index up to but not including the anchor point. If index is the same as the anchor point, do nothing. If index is after the anchor point, set the selection to the characters from the anchor point up to but not including index. The anchor point is determined by the most recent select from or select adjust command in this widget.

If the selection isn't in this widget then a new selection is created using the most recent anchor point specified for the widget.

7.15 Scale

```
type Scale = PWidget Sca
data Sca -- abstract
vscale' :: WPath -> [Conf Sca] -> GUI Scale
hscale' :: WPath -> [Conf Sca] -> GUI Scale
vscale :: Window -> [Conf Sca] -> GUI Scale
hscale :: Window -> [Conf Sca] -> GUI Scale
getScale :: Scale -> GUI Int
setScale :: Scale -> Int -> GUI ()
Options: activeforeground, background, borderwidth, command,
cursor, font, foreground, highlightbackground, highlightcolor,
highlightthickness, hor–orient, active–state, relief, sca–from,
sca–length, sca–to, sliderlength, takefocus, tickinterval,
troughcolor, wgt–label, variable, width.
```

A Scale is a slider that can be horizontal or vertical. Use getScale to get the current position. Use setScale to set the current slider position.

7.27 Listbox

```
type Listbox = PWidget Lis
data Lis -- abstract
instance ScrollableY Lis
instance Scan        Lis
listbox' :: WPath -> [Conf Lis] -> GUI Listbox
listbox :: Window -> [Conf Lis] -> GUI Listbox

insertListbox :: Listbox -> LIndex -> [String] -> GUI ()
deleteListbox :: Listbox -> LIndex -> LIndex -> GUI ()
resetListbox :: Listbox -> [String] -> GUI ()

getListboxEntries :: Listbox -> LIndex -> LIndex -> GUI [String]
getListboxSize :: Listbox -> GUI Int
listboxMoveToSee :: Listbox -> LIndex -> GUI ()

addListboxSelection :: Listbox -> LIndex -> LIndex -> GUI ()
clearListboxSelection :: Listbox -> LIndex -> LIndex -> GUI ()
setListboxSelectionAnchor :: Listbox -> LIndex -> GUI ()
getListboxSelection :: Listbox -> GUI [Int]

data LIndex = LIndex Int | LIndexActive | LIndexAnchor | LIndexEnd
            | LIndexAt ! Int ! Int | LFree String

Options: background, foreground, font, borderwidth, cursor, relief,
width, highlightbackground, highlightcolor, highlightthickness,
takefocus, height, selectbackground, selectforeground,
selectborderwidth, setgrid.
```

A listbox is a collection of list items. Items can be added, and removed from the listbox referred to by list indices, using insertListbox and deleteListbox. Individual items are Strings. The list can also be reset using resetListbox. To get all entries between two indices use getListboxEntries. To get the size use getListboxSize. To move the listbox so a specific item is visible use listboxMoveToSee. The listbox selection can be checked using getListboxSelection, the selection anchor can be set with setListboxSelectionAnchor. Any selections between two given points can be cleared using clearListboxSelection. The listbox selection can be increased with addListboxSelection.

Listbox indices are referred to by item position (LIndex), the active item (LindexActive) the selection anchor LIndexAnchor, the end of the index LIndexEnd, and at a particular pixel location (LIndexAt). To give arbitrary tcl code to use as an index use LFree.

7.28 Edit

```
type Edit = PWidget Edi
```

```
data Edi -- abstract
instance ScrollableX Edi
instance ScrollableY Edi
instance Scan        Edi
edit' :: WPath -> [Conf Edi] -> GUI Edit
edit :: Window -> [Conf Edi] -> GUI Edit

getEdit :: Edit -> GUI String
getFromTo :: Edit -> TIndex -> TIndex -> GUI String

loadEdit :: Edit -> FilePath -> GUI ()
saveEdit :: Edit -> FilePath -> GUI ()

resetEdit :: Edit -> String -> GUI ()
deleteEdit :: Edit -> TIndex -> TIndex -> GUI ()
insertEdit :: Edit -> String -> String -> GUI ()
insertEditTagged :: :: Edit -> TIndex -> String -> [TagId] -> GUI ()

eqTIndex,ltTIndex,gtTIndex :: Edit -> TIndex -> TIndex -> GUI Bool
cmpTIndex :: Edit -> TIndex -> TIndex -> GUI Ordering

searchEdit :: Edit
           -> Bool -- forward if True (back otherwise)
           -> Bool -- ignore case if True
           -> Bool -- treat pattern as regexp
           -> String -- pattern
           -> TIndex -- start at
           -> TIndex -- stop at
           -> GUI (Maybe ((Int,Int),Int))
cutClipboard :: Edit -> GUI ()
copyClipboard :: Edit -> GUI ()
pasteClipboard :: Edit -> GUI ()

data TIndex = TIndex ! Int ! Int -- Line and Char
       | TIndexAt ! Int ! Int
       | TIndexEnd
       | TIndexMark ! MarkId
       | TIndexTagFirst ! TagId
       | TIndexTagLast ! TagId
       | TIndexEmbeddedWin ! WPath

Options: background, borderwidth, cursor, font, foreground, height,
highlightbackground, highlightcolor, highlightthickness,
active—state, padx, pady, relief, selectbackground, selectborderwidth,
selectforeground, setgrid, takefocus, width, wrap.
```

An edit widget is a multiline text area. It has powerful functionality. To get the entire contents use getEdit; to get the contents between two points use getFromTo. To load a file into a widget use loadEdit; to save the edit contents into a file use saveEdit. To set the edit with a given String use resetEdit. To delete between two indices use deleteEdit. To insert use insertEdit. To compare two indices to see whether they appear before or after each other in the text use cmpTIndex. To copy, cut and paste the selection into the clipboard use cutClipboard, copyClipboard and pasteClipboard. To search a text edit for a given pattern use searchEdit.

The constructors in the TIndex data type mean the following. An index is a line and character; pixel location; the end of the text; a given mark; the start of a tag; the end of a tag; the name of an embedded widget.

Text edit areas can contain tags, marks and embedded widgets that allow access to individual text in the area.
(Note that the active_state option sets whether the widget is Read-Only/Read-Write).

7.29 Mark

```
data Mark = Mark
type MarkId = String
markId :: MarkId
markEdit :: Edit
mark' :: Edit -> MarkId -> TIndex -> GUI Mark
mark :: Edit -> TIndex -> GUI Mark
```

```
setMark :: Mark -> TIndex -> GUI ()
getMarkPos :: Mark -> GUI (Int,Int)
removeMark :: Mark -> GUI ()
setMarkGravity :: Mark -> Gravity -> GUI ()
getMarkGravity :: Mark -> GUI Gravity
data Gravity = GLeft | GRight deriving (Eq,Show)
getAllMarks :: Edit -> GUI [Mark]
getMark :: Edit -> MarkId -> Mark
previousMark :: Edit -> TIndex -> GUI Mark
nextMark :: Edit -> TIndex -> GUI Mark
insertionMark :: Edit -> Mark
currentMark :: Edit -> Mark
```

A mark is a particular location within a text area. A mark has a String Identifier (MarkId). It has a gravity. When the text it is associated with is moved it may go left or right. The insertion cursor and current mouse position are both represented by marks.

7.29 Tag

```
type Tag = TWidget Tg
data Tg -- abstract
type TagId = String
tagId :: Tag -> TagId
tagEdit :: Tag -> GUI Edit
tag' :: Edit -> TagId -> [TIndex] -> [Conf Tg] -> GUI Tag
tag :: Edit -> [TIndex] -> [Conf Tg] -> GUI Tag
putPosTag :: Edit -> TIndex -> String -> [Conf Tg] -> GUI Tag
setWithTags :: Edit -> TagId -> [TIndex] -> GUI ()
lowerTag :: Tag -> Maybe TagId -> GUI ()
raiseTag :: Tag -> Maybe TagId -> GUI ()
getAllTags :: Edit -> GUI [TagId]
getTagsAt :: Edit -> TIndex -> GUI [TagId]
tagRemove :: Tag -> [TIndex] -> GUI ()
tagRanges :: Tag -> GUI [((Int,Int),(Int,Int))]
tagNextRange :: Tag -> TIndex -> TIndex -> GUI (Maybe ((Int,Int),(Int,Int)))
tagPrevRange :: Tag -> TIndex -> TIndex -> GUI (Maybe ((Int,Int),(Int,Int)))
tagText :: Tag -> GUI [String]
selectionTag :: Edit -> GUI Tag
Options: background, borderwidth, font, foreground, justify,
relief, underline.
```

A tag is a section of text that can be modified on its own and can have actions bound to it. Tags allow hyperlinks to be produced. A tag has a particular TagId that is referred to in the edit TIndex. A tag may be moved around. The selection area in the edit is a special tag. Tags can be placed in the text using tag' and tag. A tag can cover several indexed points. These can be added with setWithTags and removed using tagRanges. All the tags in an edit area can be found with tagRanges. The text edit area can be searched backwards and forwards for tags using tagNextRange and tagPrevRange. Tags can be raised and lowered in stacking order. This is important to carry out user input on overlapping tags. To get the tags at an indexed location use getTagsAt. To get all text covered by tags use tagText.

7.30 Embedded widgets

```
type Embedded = EWidget Ew
data Ew = Ew
embedded' :: Edit -> WPath -> TIndex -> [Conf Ew] -> GUI Embedded
embedded :: Edit -> PWidget a' -> TIndex -> [Conf Ew] -> GUI Embedded
stretch :: Bool -> Conf Ew
align :: Align -> Conf Ew
data Align = AlignTop | AlignCenter | AlignBottom | AlignBaseLine
 deriving Eq
getAllEmbedded :: Edit -> GUI [WPath]
options: padx,pady
```

Embedded widgets in text areas. Place a packable widget inside a text area. When the widget is deleted from the text area then the PWidget is also destroyed.

8 State

```
type GUIRef a -- abstract
type GUIArray a -- abstract
newState :: a -> GUI (GUIRef a)
readState :: GUIRef a -> GUI a
writeState :: GUIRef a -> a -> GUI ()
modState :: GUIRef a -> (a->a) -> GUI ()
newGUIArray :: Int -> a -> GUI (GUIArray a)
readGUIArray :: GUIArray a -> Int -> GUI a
writeGUIArray :: GUIArray a -> Int -> a -> GUI ()
modGUIArray :: GUIArray a -> Int -> (a->a) -> GUI ()
```

9 Dialog boxes

```
getOpenFileName :: GUI (Maybe String)
getSaveFileName :: GUI (Maybe String)
mkDialog :: a -> GUIRef (Maybe a) -> Window -> GUI a
tcl_eventUntil :: GUIRef a -> (a -> Bool) -> GUI ()
```

To run an open and save as dialog box use getOpenFileName and getSaveFileName.

To write your own dialog boxes make use of mkDialog and the more primitive tcl_eventUntil. For instance, the following makes a modal dialog box that displays a given String. It returns the value True if ok is clicked, and False if the window is destroyed or cancel is pressed.

```
primDialog :: String -> GUI Bool
primDialog s = do
  ref <- newState Nothing

  w <- window []
  title w "Modal Dialog"
  lbl <- label w [text s]
  f <- frame w []
  packAdd lbl [packV]
  packAdd f [packAnchor C,fillX,packV,expand True]

  ok <- button w [text "Ok",command $ writeState ref (Just True)]
  cancel <- button w [text "Cancel",command $ writeState ref (Just False)]
  mapM (\w -> packAdd w [expand True,fillX,packH]) [ok,cancel]

  mkDialog False ref w
```

To see it run try Dialog.hs in the demos directory. The magic all happens in mkDialog. The definition of mkDialog is:

```
mkDialog :: a -> GUIRef (Maybe a) -> Window -> GUI a
mkDialog def ref w = do
  trapDeleteWindow w (writeState ref $ Just def)
  tcl_eventUntil (fmap isJust $ readState ref)
  mb <- readState ref
  destroy w
  return $ maybe def id mb
```

We take a default value, a GUIRef and a Window. When the window is destroyed the value Just def is placed in the GUI ref. We then run the event loop using tcl_eventUntil until the GUIRef contains a Just value. We destroy the window and then return the value from the calculation.

The function tcl_eventUntil runs the event loop processing every event until its GUI action parameter returns True, or the root window is destroyed.

10 Delayed Events

```
type Remover = GUI ()
after :: Int -> GUI () -> GUI Remover
```

11 Odds and Ends

```
type WTag = String
parseInt :: String -> Int
```

```
rgb :: (Int,Int,Int) -> String
tcl_callback :: String -> ([String]->GUI ()) -> GUI (String,GUI ())
trapDeleteWindow :: Window -> GUI () -> GUI ()
getTclTime :: GUI Double -- time in seconds since program started
tcl_debug :: Bool -> GUI () – print debugging info or not
```

The event buffer is limited to 20 slots, each 100 characters.

12 Concurrency

TclHaskell should run under concurrent haskell. It now supports pre-emptive concurrency within ghc. That is, the tcl event loop will not interfere with pre-emption in the ghc scheduler.

References

[1] J. K. Ousterhout. Tcl and the Tk Toolkit. Addison Wesley, 1994.
[2] Simon Peyton Jones, et al. Haskell 98: A Non-strict, Purely Functional Language, February 1999.
[3] B. B. Welch. Practical Programming in Tcl and Tk. Prentice Hall, 1997. Second Edition.