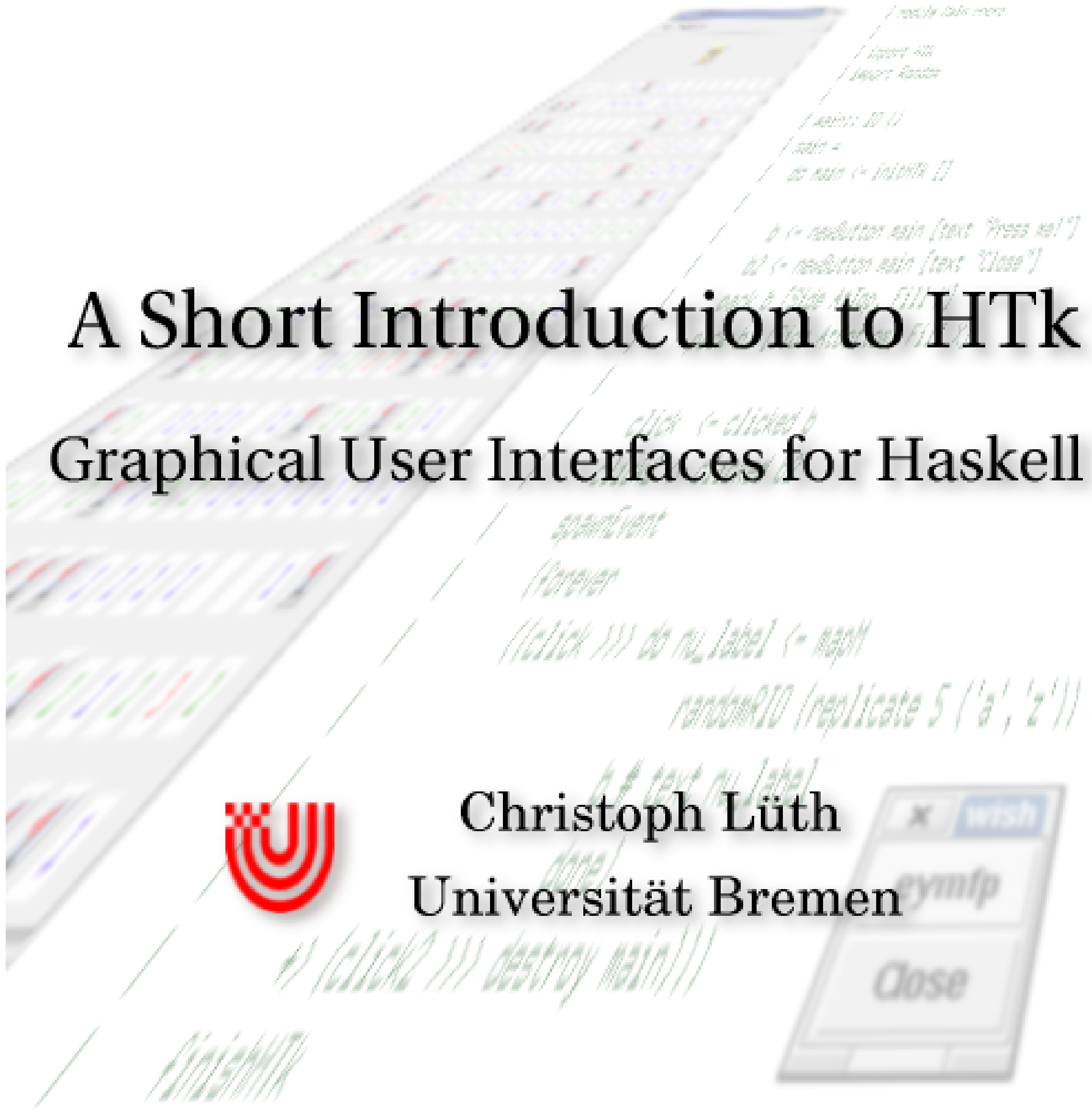


A Short Introduction to HTk

Graphical User Interfaces for Haskell

Christoph Lüth
Universität Bremen

Abstract

This paper introduces the basics of HTK, a toolkit to build graphical user interfaces in Haskell. HTK is an encapsulation of Tcl/Tk [4, 7], but we will not assume any previous knowledge of Tcl/Tk. The article is meant as a rough guide and introduction to the structure of HTK. It is not meant as a complete reference manual; Rather, it should give readers enough information and background to get them started on their first HTK programs, to know which parts of HTK might be potentially useful in the applications they have in mind, what is feasible to build with HTK, and finally to enable them to find further information quickly in the reference material.

Revision 1.3 of 2002/05/28

Contents

How to Read This Document

For maximum benefit, you should read this text in its PDF incarnation, and enable your acrobat reader to display weblinks (menu `File`, submenu `Preferences`, submenu `Weblink`). If you have configured HTK correctly, then the hyperlinks in this document should display the corresponding online reference material; this should take you to the index page. The default set-up links you directly to the HTk web-site; if you want to use a local copy of the reference manual, you should configure the Base URL in the acrobat reader (menu `File`, submenu `Document Info`, submenu `Base URL`) to point to your local copy. Finally, the sources of all the examples in this manual can found in your HTK distribution under `htk/examples/intro`.

1 Getting Started

1.1 Basics

When we design and implement a graphical user interface, we have to take two aspects into account: the *static* aspect, which specifies its appearance (which buttons to place where, what menus to display, etc.), and the *dynamic* aspect, which specifies its behaviour in reaction to the user's actions. The two can be interlinked: as a reaction to the user's action, the graphical user interface may change its appearance.

In HTK, these two aspects are modelled by *monads*. The static aspect is modelled in the `IO` monad, where all of Haskell's external interactions takes place. The dynamic aspect is modelled by *events*. For a more complete description of events, we refer to [6], but for the time being, events are an abstract datatype with three main operations:

- The central operation is *synchronisation*

```
sync :: Event a-> IO a
```

which holds the current thread until an event of type `Event a` occurs. The basic events will be user interactions, such as clicking a button; more complex events can be built using the following operators.

- The *sequencing* operators

  ```
  (>>>=) :: Event a-> (a-> IO b)-> Event b
  (>>>)  :: Event a-> IO b-> Event b
  ```

  attaches an IO action to an event; if we synchronize on this event, the IO action will be performed after the event occurs.

- Finally, the *always* operator turns an IO action into an event:

  ```
  always :: IO a-> Event a
  ```

Events form a monad, with monad composition given by `>>>=` and `always`. The monad composition corresponds to event sequencing; if we synchronise on an event composed from simple events, we wait for each of them in turn. By composing events in this way, fairly complex behaviours can be modelled in a single event.

That the dynamic behaviour is not modelled with `IO` actions directly reflects the fact that user interaction in a graphical user interface is different from other forms of I/O, because it happens asynchronously.

Further, events allow the user interface to be concurrent in a natural and controlled way, leading to a reasonable and still tractable degree of concurrency; the function

```
spawnEvent :: Event () -> IO (IO ())
```

spawns a concurrent thread which syncronises on the given event. The `IO` action returned kills the concurrent thread.

1.2 A First Example

To make this more concrete, consider a very simple example. We want to open a window which contains just one button, which should be labelled *Press me!*. Whenever the user obligingly presses the button, it should change its label to a different random string.

The static part of this program is fairly simple. There will be an initialization function (which opens the window and such), and we want to build a button with the inscription `Press me`. The following code achieves this:

```
main:: IO ()
main =
  do main <- initHTk []

     b <- newButton main [text "Press me!"]
     pack b []
```

This introduces three important concepts in HTK:

- firstly, the elements of the graphical user interface are organized hierarchically. When we create a new button, we have to pass it the GUI element which it is part of (here, the main window).
- secondly, GUI elements are created with functions called `newX`, which take a *configuration* list as argument. The configuration determines the visual appearance; here, the text which is displayed on the button. GUI elements, such as this button, are called *widgets*.
- thirdly, creating a widget does not display it *per se*. To display it, we have to explicitly place it on the screen; this is done with the `pack` command. This command takes a list of packing options as argument; more on that below.

Figure 1: A simple example.

Fig. 1 shows the result of the two operations.[1] To specify the dynamic behaviour, we need two ingredients: firstly, we need to connect the external event of the user clicking the button with an element of the data type `Event`, and secondly, we need to set up the program such that it reacts to the occurence of this event by changing the button's label.

Setting up external events to produce an `Event a` is called *binding*. When we bind an external event, we specify the external action that we wish to bind (e.g. this button being clicked, mouse movement over this window, or right button being clicked with control-key being pressed and user doing a handstand whilst whistling 'Auld Lang Syne'). The general case is the `bind` function which we will see below, but for the simple case of a button being clicked, we can use the function

```
clicked :: Button a-> IO (Event ())
```

The composed event is the click of the button, followed by changing the label. The following code achieves the desired effect:

[1]The decoration of the window — border, buttons etc — may look different on your system, since it depends on your window manager.

```
    click <- clicked b
    spawnEvent
     (forever
       (click >>> do nu_label <- mapM randomRIO
                                         (replicate 5 ('a','z'))
                    b # text nu_label))
    finishHTk
```

Here, `randomRIO (replicate 5 ('a','z'))` generates a list of five actions of type `IO Char`, and `mapM` evaluates them to a random string of length 5. The next line sets the label to this random string; how exactly this works will be explained below. Fig. 2 shows the result.

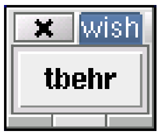


Figure 2: After clicking.

Another function requires an explanation here: `forever :: Event a-> Event a` takes an event, and returns this event composed with itself. Thus, synchronising on this event will synchronise on it once, then wait for this event to occur again. Had we left out the `forever`, our program would just wait for one button press, change the text of the button once and go on its merry way (in this case, terminate). With `forever`, we have it waiting for the next button press after the first one occurs.

As mentioned above, `spawnEvent` takes an event, and creates a concurrent thread which synchronises on this event. This is not strictly necessary here, since we do nothing else, but it is good practice to leave handling of events to threads different from the main thread. Exactly how many threads one creates — one for each button, or just one for the whole GUI — is a matter of taste and judgement.

At the end of the program, the main thread has to wait for the GUI to finish; if it just exited, the whole program would terminate. We do this by calling `finishHTk`. This also handles the case that the user closes the window by external means (e.g. the close button provided by the window manager).

Note that our program is non-terminating. If the window manager does not provide means to close a running application, we will have to use `kill` or `xkill` to stop it. This is clearly unsatisfactory, so we will now provide a second button to close the window regularly.

1.3 A Second Example

We have to augment our previous program in two aspects: statically, we have to provide another button, and dynamically, we have to react to this button being pressed by ending the program.

For the first part, we create the second button just like the first part. When we place it, we have to specify where it is going to be placed. We want it below the second button, and we want both buttons to stretch out horizontally such that they are of the same length, regardless of the size of the labels. This is done by adding *packing options* to the `pack` command. Here, `Side` says we want the first button at the top and the second at the buttom, and `Fill X` specifies the stretching behaviour mentioned above (Fig. 3 shows the resulting window):

```
main:: IO ()
main =
  do main <- initHTk []

     b <- newButton main [text "Press me!"]
     b2 <- newButton main [text "Close"]
     pack b [Side AtTop, Fill X]
     pack b2 [Side AtBottom, Fill X]
```

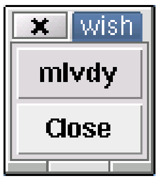


Figure 3: A second example.

To change the dynamic behaviour, we first need the second button to create an event with the `clicked` function. However, we need to change the behaviour of the spawned event such that when this new clicked event occurs, the program is finished.

This combination of events as a case distinction — when this event occurs, do something, when the other event occurs, do something different — is achieved by the fourth important operation on events, the *choice* combinator

```
(+>) :: Event a-> Event a-> Event a
```

Hence, we need to combine the previous dynamic behaviour and the new behaviour by `+>`. The new behaviour, finishing the program, is achieved by calling the `destroy` action on `main`. This closes the main window and lets the program terminate gracefully:

```
     click  <- clicked b
     click2 <- clicked b2
     spawnEvent
      (forever
        ((click >>> do nu_label <- mapM randomRIO
                                        (replicate 5 ('a','z'))
                       b # text nu_label
                       done)
        +> (click2 >>> destroy main)))
     finishHTk
```

Note that the choice occurs inside the `forever` (why?). We could also have created two threads here, each listening to one button. While in this simple situation, this would have been easier, it is in general good practice to create only as many threads as needed, since one otherwise tends to run into memory leaks by unused threads lying around or even worse, nasty synchronisation problems.

1.4 Structure of this Paper

The rest of this paper is organized as follows: we will first explain the organization of the datatypes modelling the static behaviour of the graphical user interface. In Section 3, we will describe events and in particular how to generate them from user input. After this, we will embark on a guided tour through HTK's widgets (Sect. 4 to Sect. 9), covering basic widgets, menus, editors, canvasses, windows and tix widgets, finishing off with an overview of HTK's toolkit.

2 Elements of HTK

In general, HTK has a couple of abstract datatypes used to model elements of the graphical user interface, such as buttons, menus, short text fields, longer texts, and so on. Recall from Section 1.2 above that there is an abstract datatype `Button`, elements of which are created with the function

```
newButton :: Container par=> par-> [Config Button]-> IO Button
```

This takes a parent GUI element, and a list of configurations as parameter. We will first examine the class `Container`, which models the GUI element hierarchy, and then the configurations, followed by resources such as fonts and colours. At the end of this section, we explain packing.

2.1 The GUI element hierarchy and the `Container` class.

The class `Container` designates GUI elements into which other GUI elements may be packed.

Instances of `Container` include `Toplevel` (windows), `HTk` (Tk's root window), `Frame`, and furthermore `Canvas`, and `Editor` (and a few Tix widgets).

The class `Container` is *abstract* — it has no class functions, and only serves to structure the code. Abstract classes are used frequently in HTK to impose a typing discipline onto Tk's untyped GUI element structure, with the benefit that type checking can prevent run time errors.

2.2 Configurations

Above, the text of the button was set with a *configuration option*. Configuration options determine various attributes of a widget. They can be given at the time of creation, or changed later on. In general, the configuration type is just a type synonym[2]

```
type Config w = w -> IO w
```

As seen above, configurations can be given at the time of creation, or later on. In the latter case, the helpful `(#)` operator provides useful syntactic sugar:

```
( # ) :: a -> (a -> b) -> b
o # f = f o
```

Some configurations are only supported by one particular widget, and thus are a simple monomorphic function. However, most configurations are supported by many, but not all widgets, and each configuration by different ones, and this behaviour is modelled in HTK by type classes. Common configuration classes can be found in the module `Configuration`. For example, the text configuration is given by this class:

```
class (GUIObject w, GUIValue v) => HasText w v where
  text :: HasText w v => v -> Config w
  getText :: HasText w v => w -> IO v
```

The class `GUIObject w` is one of HTK's most basic classes. Its instances are widgets and other interface elements we will encounter later (canvas items, text tags, windows). `GUIValue v` is another basic class, the instances of which are all basic datatypes which can be communicated to Tk: `Int`, `Double`, `Bool`, `String` and `[String]`. Now, all widgets can be configured with a text are instances of the class `HasText`, such as `Button`.

2.3 The class `Destroyable`

The class `Destroyable a` has the main class function

```
  destroy :: a-> IO ()
```

which when called will destroy its argument. All GUI elements are instances of this class, and calling the destroy function will remove them from the screen. In particular, destroying the return value of `initHTk` will close all windows opened by the program (i.e. remove the whole GUI).

[2]Type synonyms like that in class confusions confuse Hdoc, which is why they appear expanded at various places of HTK's source code— just in case you happen to browse it, which you are more than welcome to.

2.4 Concurrency and the class `Synchronize`

The class `Synchronize a`, with its main and only class function

```
  synchronize a :: a-> IO b-> IO b
```

describes a *monitor*, like the synonymous method in the programming language JAVA [1]. All GUI elements are instances of this class. It allows convenient handling of concurrency— if two IO actions must not interleave, you synchronize them on any one GUI element, typically the one generating the event which the function calling the two IO actions is bound to.

2.5 Resources

Resources are auxiliary datatypes modelling distances, colours, fonts, pictures and cursor shapes.

2.5.1 Geometry

The abstract data type `Distance`, implemented in the module `Geometry`, represents distances in HTK. Distances can be specified in `cm`, `mm`, `ic` (inches) and `pp` (points), with functions `cm:: Int-> Distance` etc. Moreover, `Distance` is an instance of `Num`, so we can specify the distance 3 (meaning 3 pixels) directly.

2.5.2 Colours

The abstract data type `Colour`, implemented in the module `Geometry`, represents colours in HTk. Just like distances, the type itself is abstract, but unlike distances, there is a class `ColourDesignator` allowing colours to be specified in a flexible manner by overloading. Functions expecting a colour as an argument take any instance of `ColourDesignator` as argument, such as

```
background :: (ColourDesignator c, HasColour w) => c -> Config w
foreground :: (ColourDesignator c, HasColour w) => c -> Config w
```

which set the background or foreground of any interface element which has a colour (nearly all). Its main instances are:

```
instance ColourDesignator [Char]
instance ColourDesignator (Int, Int, Int)
instance ColourDesignator (Double, Double, Double)
```

The strings are named colours (`red`, `white`, `black`, etc.), the tuples are RGB values. (The functions of the type classes `HasColour` and `ColourDesignator` are for HTk's internal consumption only, but annoyingly appear all over the documentation.)

2.5.3 Fonts

Fonts are implemented in the module `Font`. The datatype `Font` represents the specification of exactly one font in the usual (X-style) way, by giving a family, slant, spacing, width and weight. For example, the family is given by

```
data FontFamily = Lucida | Times | Helvetica
                | Courier | Symbol | Other String
```

where the five enumerated types are available on most systems. With `Other`, you can directly give a more exotic family such as `clearlyu alternate glyphs`.

Just like with colours, there is a class `FontDesignator`, the instances of which give ways of describing fonts, such as the following:

```
instance FontDesignator FontFamily
instance FontDesignator (FontFamily,Int)
instance FontDesignator (FontFamily,FontWeight,Int)
instance FontDesignator (FontFamily,FontSlant,Int)
instance FontDesignator String
instance FontDesignator XFont
```

The first three allows fonts to be specified by the font family, plus the size, plus the weight or the slant, respectively; the second from last allows an X-style string (e.g. `-adobe-courier-bold-o-normal-*-14-*-*-*-*-iso8859-*-`), and the last an X-style specification as an abstract datatype.

Be warned that fonts are, in principle, not very portable under X, since the available fonts are determined by the fonts of the X server the programm is running on. It is best to stick to well-known font families such as the above, and usual sizes. (So, no `clearlyu alternate glyphs` in 144 pixels.)

2.5.4 Images and Bitmaps

Images are representation of pictures, from wee icons to screen-filling murals. HTK's images are what in Tk is called a photo. An image is created with the function

```
newImage :: [Config Image] -> IO Image
```

Once created, an image can be attached (via the `photo` configuration, class `HasPhoto`) to labels, buttons, and canvas items. The configuration gives the image's data, either via the `filename` configuration, where a file is specified which contains the image data, or via the `imgData` configuration, which passes the data directly as Base64 encoded string. (You can use e.g. the `mimencode(1)` utility to produce such a string.) As can be seen from the `Format` datatype, HTK supports GIF, PPM and PGM formats. The latter method of giving the image data is preferable, since it makes the compiled executable independent of image data files; if you pass a filename, you need to make sure that the image file is to be found at that path during the *runtime* of the program!

Bitmaps have a foreground and a background. The difference between images and bitmaps is that bitmaps are more versatile, e.g. you can change the foreground and background, but only have two colours. There is also a number of predefined bitmaps (`Hourglass, Questhead`, etc.) Apart from that, bitmaps are rather boring.

2.5.5 Cursors

Any widget can change the shape of the cursor, i.e. the shape displayed by the cursor while over this widget, by using the `cursor` configuration. The class `CursorDesignator` from the module Cursor module allows cursors to be specified in an overloaded way (like colours and fonts). The simplest way to get a cursor is to avail yourself of some of the predefined ones (at least under X windows), but you can build your own cursor by giving a bitmap, a mask, and colours.

2.6 Packing

As mentioned above, after widgets have been created (with e.g. `newButton`), they will not be displayed just yet; this only happens after they have been packed. One can use this effect by first creating lots of widgets, and then packing them in one go, lessening the unpleasant flicker effect occuring when the GUI is built one interface at a time. To minimize the flickering further, use the function

```
delayWish :: IO a-> IO a
```

which delays all `Tk` commands in the argument, and performs them at once. The argument then consists of a sequence of `pack` commands. Unfortunately, the flickering effect cannot be totally eliminated.

Packing in particular determines the visual layout of the GUI by the order in which the widgets are packed, and by packing options. Tk's know different packing algorithms (or *geometry managers*, in Tk parlance); of these, HTK supports the pack geometry manager, and the grid geometry manager. The third, the place geometry manager, could easily be added, if you are so inclined.

2.6.1 The Standard Packer

The behaviour of the standard packer is easily explained, and hard to understand. Widgets are packed with the function

```
pack::Widget w => w -> [PackOption] -> IO ()
```

The datatype PackOption is defined as

```
data PackOption = Side SideSpec  | Fill FillSpec
                | Expand Toggle  | Anchor Anchor
                | IPadX Distance | IPadY Distance
                | PadX Distance  | PadY Distance
```

The first two constructors are most important here. The SideSpec specifies where the widget is packed (top, bottom, left, right), and FillSpec specifies in which direction it expands to fill the available space. Bear in mind that widgets are packed as tight as possible, and that once packed, they are never repacked, moved or resized. That is, if e.g. a widget is packed against the top, it will sit in the middle (if no `Fill X` is specified), and will not move if a widget is packed against the right-hand side, even if the window is increased in size to make a new widget fit.

`Expand` just means that the widget expands when the containing element is expanded (i.e. the window is resized), and `Anchor` specifies a gravity (a side to which the widgets stick). The rest create a padding border around the widget in various directions.

It is quite normal that most of the time the arrangement of the widgets will not look like intended, and you will need to use frame widgets (see Section 4.1).

2.6.2 The Grid Packer

The grid packer divides the container widget into a grid, and allows placement of widgets relative to that grid. To pack a single widget use the following function:

```
grid :: Widget w => w -> [GridPackOption] -> IO ()
```

The datatype `GridPackOptions` specifies the packing options for the grid packer.

```
data GridPackOption =
    Column Int | Row Int | GridPos (Int, Int)
  | Sticky StickyKind | Columnspan Int| Rowspan Int
  | GridPadX Int | GridPadY Int | GridIPadX Int | GridIPadY Int
```

Within the same container you cannot use different packing algorithms. The first widget packed into a container defines the packing for this container.

3 Events

In general, events are an abstract datatype for communication and synchronisation, much in the spirit of process algebras such as CCS [2], CSP [5] or the π calculus [3]. Here, an `Event` is an abstract datatype with operations such as `sync`, `+>` and `>>>=`, which additionally form a monad; we refer to [6] for more information.

In HTK, basic events arise from user interactions by means of the *bind* commands. By binding a user interaction (such as clicking a button), we set it up to produce an event, on which we can synchronise and thus produce a reaction to the user's action. We can then use the event operators to build more complex events with these basic events.

One important caveat here is that once you set up a binding, you *must* eventually *synchronise* on the resulting events, since otherwise the unused events will pile up and result in a memory leak.

Bindings are generated by calling one of `clicked`, `bindSimple` and `bind`, where `bind` is most flexible, as can be seen by its complex type.

3.1 Simple Clicks

Simple interface elements such as buttons and menu entries which are instances of the class `HasCommand`. For these, we have a function

```
clicked :: HasCommand w => w -> IO (Event ())
```

The event here occurs when the element is clicked. The event does not have any additional information — a click is just a click, after all.

3.2 Simple Binds

For all GUI elements, the function `bindSimple` provides a simple way of binding. Its arguments are the GUI element concerned, and a specification of the kind of events we are interested in:

```
bindSimple::GUIObject wid => wid -> WishEventType
                                 -> IO (Event (), IO ())
```

WishEventType is an algebraic data type describing the kind of event we would like to bind to, much along the lines of Tk's events:

```
data WishEventType =
   ButtonPress (Maybe BNo) | ButtonRelease (Maybe BNo) |
   Motion |
   Enter  | Leave |
   KeyPress (Maybe KeySym) | KeyRelease (Maybe KeySym)|
   Activate | Circulate | Colormap | Configure | Deactivate |
   Destroy | Expose | FocusIn | FocusOut | Gravity |
   Map | Property | Reparent | Unmap |
   Visibility deriving (Show,Eq,Ord)
```

Each constructor corresponds to a different event, such as

- mouse buttons pressed and released (`ButtonPress (Maybe BNo)` and `ButtonRelease (Maybe BNo)`, where `BNo` is just `Int`),
- mouse movements (`Motion`),
- the mouse entering or leaving a GUI element (`Enter`, `Leave`),
- keys pressed or release (`KeyPress (Maybe KeySym)`,`KeyRelease (Maybe KeySym)`),
- and various window events (`Map`, `Unmap`, `Expose`, `Activate`,...).

Note that not every GUI element can generate every event. Obviously, only windows can generate window events, but more subtly, only entry widgets, editor widgets and toplevels (i.e. windows) can generate `Key` events.

The return value of `bind` is an event, and an IO action which unbinds the event. You should use this action if you are not interested in the event anymore (i.e. it will not be synchronised on anymore), otherwise there will be a memory leak.

3.3 Full Bindings

In fact, `bind` is only a simplified version of the function bind

```
bind::GUIObject wid => wid -> [WishEvent]
                           -> IO (Event EventInfo, IO ())
data WishEvent = WishEvent [WishEventModifier]
                           WishEventType
```

where `WishEventModifier` desribes possible event modifiers such as `Shift`, `Alt`, `Meta` (corresponding to the Shift, Alt or Meta key being held), `Button1`, . . . , `Button5` (corresponding to mouse button 1 thru 5 being pressed) or `Double` and `Triple`. Again, not all combinations of modifiers and events make sense; for example, double and triple pertain to mouse button presses, and modifying a mouse button press with a different button is not really helpful. Note that the argument of bind is a list of events, meaning that the interface events have to occur in (rapid) sequence to generate the event.

Here, the first component of the return value is an event of `EventInfo`, which is a labelled record as follows:

```
data EventInfo = EventInfo { x :: Distance,
                             y :: Distance,
                             xRoot :: Distance,
                             yRoot :: Distance,
                             button :: Int }
```

The information here is the `x` and `y` component of the mouse position, both relative to the window in which the event occurs, and the root window (i.e. the screen), and the button being pressed.

4 Basic Widgets

In the following sections, we give a brief tour around HTK's widgets. We have simple widgets which are explained in short sentences of words with less than three syllables, menus, text widgets, and canvasses, which require a slightly longer explanation.

There are more widgets: if you can use Tix (an enhance version of Tcl/Tk; you should consider using it anyway, it looks slightly less clunky than plain Tcl/Tk), there are the useful tab and pane widgets. And because HTK is an abstraction on top of Tk, there are a variety of so-called *mega-widgets* which are implemented in Haskell; these can be found in the toolkit (Sect. 10).

4.1 Frames

A frame is just an invisible container into which we can pack other widgets, grouping them together. The use of frames is in packing: the grouping is necessary most of the time to achieve the desired layout with the standard packer.

Moreover, when using a frame, we can use a different packer (see Sect. 2.6) than in the parent widget; for example, we can use the grid packer inside a window using the standard packer.

4.2 Boxes

Boxes are mega-widgets, implemented using frames. They allow simple horizontal or vertical packing (widgets are placed in the box side by side or on top of each other), and be flexible or rigid (expand when the parent window is resized, or not); e.g. `newVBox` creates a rigid box with vertical packing, and `newHFBox` creates a flexible box with horizontal packing.

4.3 Buttons

A button is a simple widget which can be clicked. Buttons can have bitmaps, images (class `HasPhoto`), or text as labels.

4.4 Labels and Messages

A label is a simple widget for text, bitmaps or images. Messages are slightly more sophisticated widgets for longer text messages; messages linebreak the text. For both, the font of the text (class `HasFont`) and the justification (class `HasJustified`) can be specified. The `aspect` configuration (for messages only) specifies the width of the text as a percentage of the height; so `aspect 200` gives a message twice as wide as high.

4.5 Entry

An *entry box* is a box which contains one editable line of text. If is used to input short texts, such as a name or a credit card number.

As opposed to previous widgets, entries are polymorph over the type of values they are supposed to hold and edit, hence an entry is created with

```
newEntry :: (Container par, GUIValue a) =>
            par -> [Config (Entry a)] -> IO (Entry a)
```

The current state of the input is the *value* of the entry; it is accessed and set with the configuration from the class `HasValue`. Note that the values need not be strings, but must be an instance of `GUIValue`.

Tk only provides the basic editing functions for entry widgets. If you want anything more, for example to read the value of the entry when the return key is pressed, you have to do this yourself by binding to the `KeyPress` event. Here is a very basic example of how to use an entry widget: we build an entry widget, and when the return key is pressed, we change the window title to the entered text, and clear the entry widget.

```
main =
  do main <- initHTk [text "Entry example"]

     f <- newFrame main []
     l <- newLabel f [text "Rename: "]
     e <- (newEntry f [value "", width 20])::IO (Entry String)

     (entered, _) <-
       bind e [WishEvent [] (KeyPress (Just (KeySym "Return")))]

     pack f []
     pack l [PadX 10, Side AtLeft]
     pack e [PadX 10, Side AtRight]

     spawnEvent
      (forever
        (entered >>> do txt <- (getValue e) :: IO String
                        e # value ""
                        main # text txt >> done))

     finishHTk
```

An alternative to using values is to use `TkVariables`. These are variables on the Tk side, which can be directly connected to the widget in the sense that the variable always holds the entry's state. The advantage of this approach is that we can share values across widgets (see `Mainhello3.hs` in `htk/examples/simple`).

Finally, entry widgets implement the quite flexible indexing and selection classes (see Sect. 6.1 and 6.2 below).

4.6 Scrollbars

I assume the esteemed reader has already seen a scrollbar. In HTK (and Tk), once you created a scrollbar you will need to connect it to the widget you want to scroll. This is done with the class HasScroller: you create a scroll bar, and attach it to the scrollable widget with `scrollbar` option. The example below shows how. Don't forget to pack both the scrollable widget and the scrollbar.

4.7 Listboxes

A listbox has a list of several items, from which you can select one. Listboxes can be scrollable, which is most helpful since they can hold more items than are visible at a given time.

Just like entries, list boxes are a polymorphic type, created with the function `newListBox` with the by now familiar signature. As opposed to entries, list boxes (quite obviously) have lists of values:

```
instance (GUIValue a, GUIValue [a]) => HasValue (ListBox a) [a]
```

When the user selects something from the the listbox, Tk's selection is set, which can be queried with the methods of the module `Select`, in particular `getSelection`. No event is generated *per se* — if you want that, you need to bind the left mouse button (which generates the selection).

Table 1 shows a short example which demonstrates the usage of listboxes, selections and scrollbars. Note the type constraint on the `newListBox` — we need this to force the type to `String`, since it can not be inferred. Fig. 4 shows a screenshot.

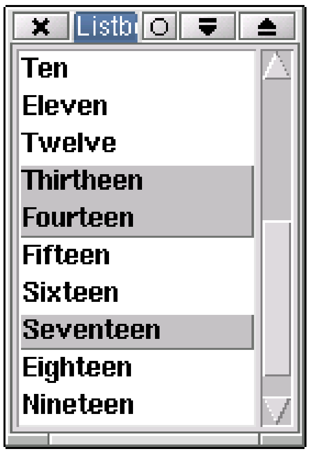


Figure 4: A list box.

The position of entries in a list box can be indexed with instances of the class `HasIndex`. For more on indices, see Section 6.1, but the important instances here are

```
instance HasIndex (ListBox a) Int Int
instance HasIndex (ListBox a) EndOfText Int
instance (Eq a, GUIValue a) =>
         HasIndex (ListBox [a])
                  (ListBoxElem a) Int
```

In other words, the index is a number (starting with 0), the `EndOfText` (only constructor of the synonymous data type), or the element itself.

A configuration particular to list boxes is the *selection mode*. The `SelectMode` is an enumeration of four constructors, which determine the way elements are selected in a list box:

- `Single` means a single element can be selected;
- `Browse` means a single element can be selected, and the selection can be dragged with the mouse;

```
module Main (main) where

import HTk
main :: IO ()
main =
  do main <- initHTk [text "A Listbox"]
     lb  <- newListBox main [value numbers, bg "white",
                            size (15, 10)] :: IO (ListBox String)
     pack lb [Side AtLeft]
     scb <- newScrollBar main []
     pack scb [Side AtRight, Fill Y]
     lb # scrollbar Vertical scb
     (press, _) <- bindSimple lb (ButtonPress (Just 1))
     lb # selectMode Extended
     spawnEvent (forever
       (press >> always
          (do sel<- getSelection lb
              putStrLn ("Selected "++
                        show (sel:: Maybe [Int])))))
     finishHTk where
  numbers =
    ["One", "Two", "Three", "Four", "Five", "Six", "Seven",
     "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirtheen",
     "Fourteen", "Fifteen", "Sixteen", "Seventeen",
     "Eighteen", "Nineteen", "Twenty"]
```

Table 1: Example program for list boxes and scrollbars.

- `Multiple` means more than one element can be selected by toggling the selection state (so to select three elements you have to select each of them in turn);
- `Extended` means more than one element can be selected, with the first selection forming the so-called *anchor*; shift and first button selects the range from the anchor to that entry; and control and first button toggles the selection state of single items. (This is the most common behaviour of list boxes in other GUI toolkits.)

Selections are handled by the selection classes (see Section 6.2) below. You can set the selection, or query the current selection as in the code above.

5 Menus

Menus are modelled by the type `Menu`. Rather surprisingly, Menus need *not* be packed. Menus can either be the normal things which appear in a menu bar (typically at the top of a window), or attached to a single button, or pop-up menus which pop up out of nowhere. Actually, these are typically considered bad interface design, since they rarely confirm to user expectations (i.e. the user has no way of knowing wether a pop-up menu will appear). Try to use pop-up Menus only as optional, convenient short-cuts. While we're at it, try also to use one menu bar only, at the top of the window, since several menu bars in one window will be confusing, and try not to change the menus while the user isn't looking.

Menus in HTk can contain:

- simple commands (which are instance of the `HasClicked` class),
- sub-menus (cascades),
- checkbuttons (boolean toggles),
- radiobuttons (multiple exclusive selections),
- and separators.

Checkbuttons and radio buttons have a state, which can be access by attaching a *Tk variable* to them (class `HasVariable`; a `TkVariable` is created with `createTkVariable`).

Here, we construct a typical menu. We start with creating a window and all that:

```
module Main (main) where
import HTk
```

```
main :: IO ()
main =
 do main <- initHTk [text "Menus!", size(300, 300)]
```

Now, we create the element holding the menu, and attach it as a menubar to the window. The second parameter to `createMenu` is a boolean determining wether this is a tear-off menu[3] (i.e. a menu which you can tear off from its menu button, and keep open).

```
    menubar <- createMenu main False []
    main # menu menubar
```

Each pull-down menu in the menubar is in fact a submenu; we first create such a submenu, and then attach a menu to it:

```
    pd1 <- createMenuCascade menubar [text "First Menu"]
    m1 <- createMenu menubar False []
    pd1 # menu m1
```

The utility function `createPulldownMenu` has been provided, since creating a pull-down menu is such a common task; it does exactly what the previous three lines have done, and will be used in the rest of this wee example. Now we create the three simple menu items, followed by a separator and a submenu of two more items:

```
    c1 <- createMenuCommand m1 [text "First Menu Point"]
    c2 <- createMenuCommand m1 [text "Second Menu Point"]
    createMenuSeparator m1 []
    s   <- createPulldownMenu m1 [text "Submenu"]
    c31 <- createMenuCommand s [text "First Subpoint"]
    c32 <- createMenuCommand s [text "Second Subpoint"]
```

Now we create a second pulldown menu with the a check button, and a group of three radio buttons. Note that Tk variables can hold all types which are instances of the class `GUIValue`, in particular strings and characters (and note how we have to disambiguate the overloaded numerals).

```
    m2 <- createPulldownMenu menubar [text "Buttons"]
    v1 <- createTkVariable True
    c1 <- createMenuCheckButton m2 [text "I am cool", variable v1]

    createMenuSeparator m2 []
```

[3]This should be a configuration, but for tedious technical reasons isn't.

```
    v2 <- createTkVariable (0::Int)
    r1 <- createMenuRadioButton m2 [text "No milk or sugar",
                                     value (0::Int), variable v2]
    r2 <- createMenuRadioButton m2 [text "Milk, no sugar",
                                     value (1::Int), variable v2]
    r3 <- createMenuRadioButton m2 [text "Sugar and milk",
                                     value (2::Int), variable v2]
```

This code in itself defines the menu. To query the state of the variables, we bind the check button, and spawn a thread which reads the variables. However, you do not need to bind anything to a button if you do not want to react on it being pressed, and just want to read the value of the variable at some point. A menu command is an instance of class `HasCommand`, so you can use `clicked` to bind a simple event to it (see Sect.).

```
    cl <- clicked c1
    spawnEvent (forever (
         (cl >>> do val1 <- readTkVariable v1
                    val2 <- readTkVariable v2
                    putStrLn ("v1: "++  show val1++
                              ", v2: "++ show val2))))
    finishHTk
```

More exciting examples of using menus— in particular pop-up menus— can be found in `htk/examples/simple/Mainmenu.hs`.

The reader will probably agree that this is a lot of code for such a simple task, and quite unnecessarily so. For the convenient design of menus, we recommend the menu modules from the toolkit (see Sect. 10.4), which are at an abstraction level more suitable for a functional language.

6 The Editor

An editor (also called a text widget) is a very flexible and powerful widget to display and edit texts. As opposed to labels and messages, is scrollable (i.e. an instance of class HasScroller), but its chief difference to labels and messages are that it can be *edited* and has *text tags*, which make into one of Tk's most powerful widgets.

Editors are created with `newEditor`. The datatype `Editor` is monomorphic, but editors have a value (their textual content), and are thus members of the class `HasValue`.

Besides the usual, editors provide configurations to

- set the spacing with the methods of the class `HasLineSpacing`,
- set the wrap mode (character, word, or no wrap) with the function `wrap`,
- and set tabulators with the class `HasTabulators`.

Text can be appended (`appendText`) or inserted at a specified position into the editor with the function `insertText`:

```
insertText::(HasIndex Editor i BaseIndex, GUIValue a) =>
            Editor -> i -> a -> IO ()
```

The position where the text is inserted is specified by the class `HasIndex`.

6.1 Indices

The type `BaseIndex` also models indices into list boxes, entry widgets *etc.*, but for editors only the constructor `IndexPos` is important:

```
data BaseIndex =
  IndexNo Int | IndexPos Position | IndexText String
```

The class `HasIndex` models which way of indexing a position inside a widget is valid.[4] For text widgets, we have

```
instance HasIndex Editor BaseIndex BaseIndex
```

so we can just specify the index as a position (i.e. a row and column number, starting from 0), but we can also specify the index by the constant `EndOfText` (only constructor of the data type `EndOfText`), or by its screen coordinates (`Pixels`), or by *modifying* one of the other indices:

```
instance HasIndex Editor EndOfText BaseIndex
instance HasIndex Editor Pixels BaseIndex
instance HasIndex Editor i BaseIndex =>
         HasIndex Editor (i, [IndexModifier]) BaseIndex
instance HasIndex Editor i BaseIndex =>
         HasIndex Editor (i, IndexModifier) BaseIndex
```

Index modifiers (only for editors) are modelled by type `IndexModifier`, and specify things like ‘three words forward’, or a ‘one line back’.

[4]From an abstract point of view, the type class is unfortunate, since the last type parameter is only for internal consumption; it models the so-called base index of the widget, i.e. the type into which other ways of indexing will have to be translated. As a user, you can just disregard the last parameter of the class.

6.2 Selections

In an editor (and an entry widget), text can be *selected* by marking it with the mouse button. Similarly, in list boxes, entries can be selected. Abstractly, these kind of selections are modelled with the selection type classes (module `Selection`). There are five classes, of which `HasSelection`, `HasSelectionIndex` and `HasSelection-IndexRange` let you set the selection, and `HasSelectionBaseIndex` and `Has-SelectionBaseIndexRange` let you query the selection. The selected text, entries *etc.* are indexed with indices (see Sect. 6.1). The `Range` classes have a pair of functions, which set or get the start or end of the selection, respectively, whereas the `HasSelectionIndex` and `HasSelectionBaseIndex` classes get or set the selection as list of indices.

This selection is Tk-internal (i.e. only works within the same wish), and should not be confused with selection as implemented by X Windows. This selection is handled by the class `HasXSelection`. Its instances are entries and editors. For those, Tk's selection can also be the X selection. The module `XSelection` allows access to the X selection (get the current selection, selecting wether widgets export their selection etc.)

6.3 Text Tags

A text tag is a way of marking a specified section of the text. One text tag can mark a list of ranges, which are specified by a start and an end index:

```
createTextTag ::
    (HasIndex Editor i1 BaseIndex,
     HasIndex Editor i2 BaseIndex) =>
    Editor -> i1 -> i2 -> [Config TextTag] -> IO TextTag
```

Once we have a text tag, we can configure it. It has a font, colour, cursor, justification, line spacing, wrapping, tabulators and so on. Moreover, you can bind events to a text tag, making it active; so when the mouse moves over the text, or the user clicks the text, something exciting happens. This allows you very easily to implement hypertext, references and so on.

When you insert into an editor, the text tags are accommodated appropriately; i.e. if you insert into the middle of a text tag, then the end of the tag is moved accordingly.

6.4 Editing

As mentioned above, the editor's text can be edited, i.e. the user can type text into the widget. This feature can be considered as a conveniently set up set of key bindings (e.g. the key 'a' is bound to the function which inserts an 'a' at the current cursor position). You can switch off the editing behaviour with the function `disable` from the class `HasEnable`, but be warned: *You can only change the text of enabled editors!* Trying to insert text into a disabled editor is one of the most common Tk errors; it will not generate a run-time error but just silently fail to change the text of the editor, leaving the programmer in puzzled dismay.

The example `htk/examples/simple/Maintexttag.hs` shows what you can do with text tags, and how to use them. The module `MarkupText` from the toolkit allows a more abstract approach to displaying text with differents fonts, hyperreferences and so on (see Section 10 below).

6.5 Marks and More

To keep track of positions within an editor, you can use marks (type `Mark`).[5] Essentially, you can put a mark into an editor at a specified position (with `createMark`), and query its position later (`getBaseIndex`). The gravity of the mark specifies wether when text is inserted at the exact position of the text, it should move to the left, or the right.

There's more you can do with an editor. For one, you can also insert widgets into the text (this is called an *embedded window*), with the function

```
createEmbeddedTextWin ::
    (HasIndex Editor i BaseIndex, Widget w) =>
    Editor -> i -> w ->
    [Config EmbeddedTextWin] -> IO EmbeddedTextWin
```

so you can have buttons, entries, and even another editor inside an editor (see the module `EmbeddedTextWin`).

Further, you can search the text inside an editor (with the `search` function), read and write the contents to and from a file, scroll the contents to a specific position, and much more as found in the module `Editor`.

7 The Canvas Widget

A canvas is a widget to draw on, i.e. we can put graphical objects on it (lines, polygons, circles and ellipses, wee images, and even whole widgets) at a specified posi-

[5]The first one to send us an email saying 'Shouldn't this type have been renamed to Euro?' will be forcefed Perl scripts until his mind explodes into a regular expression. You have been warned. Other than that, we strongly encourage any comments about HTK or this wee manual. Go on, send us that mail!

tion. A canvas is scrollable, so its drawing area can be far larger than the displayed area.

The graphical objects on a canvas are called *canvas items*. Just like for widgets, the different kinds of canvas items are represented by different types. Canvas items can be configured (but of course the configurations are mostly different from widget configurations), and we can bind events to them.

7.1 Canvas Item Configurations

The most important configurations, provided by all canvas items, are:

- `HasCoords` sets or returns the coordinates of the canvas item on the canvas. Nearly all canvas items (except for lines) have a position (`HasPosition`), and many have a size (`HasSize`).
- `FilledCanvasItem` allows to configure the filling colour, outline colour, stipple and width of the outline of the canvas item. Note that for some canvas item (e.g. lines), setting the outline does nothing, but setting the filling colour sets the colour of the item. A stipple, incidentally, is a bitmap mask used to draw the item.

7.2 Canvas Items

The following are the types of canvas items:

- `Arc`: a section of an oval, given by the start angle and the length (configurations `start` and `extent`).
- `BitmapItem` and `ImageItem`: canvas items given by bitmaps and images. The image of an image item is set with the `HasPhoto` class, just like for some widgets.
- `LineItem`: a line item is given by a list of two or more points, which define a list of line segments. The segments can be joined in various ways (with the `joinstyle` configuration), you can draw splines (class `SegmentedCanvasItem`), and put arrows on either (or both) ends (configurations `arrowshape` and `arrowstyle`).
- `Oval`: an oval is given by defining its *bounding box*, i.e. the smallest rectangle enclosing it, using the classes `HasGeometry` or `HasPosition` and `HasSize`. If the bounding box is a square, the oval is a circle.[6]

[6]Except it usually won't *look* like a circle on the screen, because on most screens pixels are rectangularly-shaped.

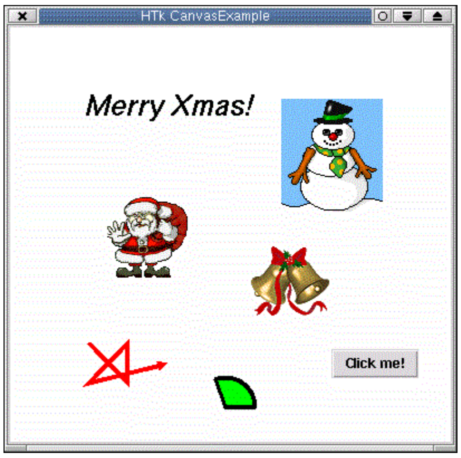


Figure 5: Canvas items: a text item, three image items, a line item, an arc and an embedded window (a button).

- `Polygon`: a polygon is given by a list of vertices. Strangely, one cannot configure the join style of the polygon, but you can configure splines.
- `Rectangle`: a rectangle is given by two points.
- `TextItem`: a text item is a text, possibly consisting of more than one line. It is an instance of `HasFont`. (Note that HTK curently does not support Tk's selection and insertion methods for text items; it should be easy to add, if you really require that.)
- `EmbeddedCanvasWin`: an embedded canvas window is a widget embedded into a canvas. All instances of class `Widget` can be embedded into a canvas. Note that you don't need to pack embedded widgets; they appear as soon as the embedded canvas window is created.

7.3 Canvas Tags

A *canvas tag* is a convenient way to refer to canvas items. A canvas tag is an instance of the class `CanvasItem` (but setting e.g. its filling colour will of course have no effect). Canvas tags are in particular useful because you can not only add canvas items explicitly to the tag, but also by the so-called *search specification*, so you can select the canvas item closest to a given canvas location etc.

For more on canvasses, see the examples under `htk/examples/canvas`. For example, `MainxmasCanvas.hs` (screenshot Fig. 5) shows a variety of canvas items, and how to set up a cheap and cheerful drag-and-drop.

8 Windows

What is commonly known as a window is called a *toplevel* widget in Tk and HTK. They are created with a function

```
createToplevel :: [Config Toplevel] -> IO Toplevel
```

but (of course) they need not be packed, they just appear. Toplevels are containers (instances of `Container`), so you can just pack other widgets into them. They are closed by destroying them (see Sect. 2.3).

The familiar properties expected from a window are all in the class `Window` from module `Window`. Its size can be configured either by classes `HasGeometry` or `HasPosition` and `HasSize`, and its title by the class `HasText`. The `screen` configuration sets the screen the window appears on (the display in X windows speak). Toplevels can have a menu attached which is the *system menu* for this window.

8.1 Window Managers

Windows are managed by the *window manager*. It controls the size and location of the windows appearing on the screen, trying to mediate between the slew of applications vying for the user's attention.

This means that even though the program can request a particular geometry (i.e. size and location), it is not guaranteed to get it; in particular, the window manager may decide to place the window somewhere else entirely.

HTK does not support all of Tk's commands to communicate with the window manager, but most of the important ones. These are the class functions of the `Window` class, as mentioned above, and the `Screen` module, which allows you to query the properties of the display (e.g. height and width).

8.2 Focus

The *focus* determines which window receives keyboard events. It is managed by the window manager. Sometimes, it is necessary for an application to *grab the focus*, i.e. require that all input go to this window before proceeding. For example when implementing a window which asks the user to confirm a choice, you do not want to user to start any other interaction, you want them to answer the question. Such a window, which has to answered before the program is allowed to proceed, is called *modal*.

Grabbing the focus is done with the functions `grabLocal` or `grabGlobal` from the module `Focus`. A local grab means that the user cannot interact with any other window from the same application; a global grab means that the use cannot interact with any other window. After grabbing the focus, you should return the grab with `returnGrab`.

Global grabs are dangerous, because if you forget to return the grab (or even worse, your application crashes), the user is effectively frozen out. On the other hand, for example when requiring sensitive information such as passwords or credit card numbers, they are a good idea to keep the user from typing his password into an open chat window.

The toolkit (see Sect. 10) offers the modules `ModalDialog` and `DialogWin` which implement modal dialogs, and various oft-used dialog windows.

9 Tix and Tix Widgets

Tix (the Tk Interface eXtension) is an enhanced version of Tcl/Tk; in particular, it replaces Tcl/Tk's rustic default looks by something a bit more modern, and it has a lot more widgets built in.

To use tix, you pass the compiled program the command line option `--uni-wish=tixwish` (where `tixwish` is the name of Tix' wish; try `which tixwish` at the command line, and if that says command not found in whatever language your shell is configured to talk to you to, you are out of luck I am afraid.) Thus, the source code does not change; Tix is a *conservative extension* of Tcl/Tk.

To find out whether the HTK program is currently running under a Tix wish, use the function `isTixAvailable:: IO Bool`; a return value `True` indicates Tix.

Here is an overview over the Tix widgets encapsulated by HTK:

- The class `HasTooltip`, instantiated by all widgets, allows tooltips: text which is attached to the widget, and appears in a wee yellow box if the mouse hovers over the widget for a short while. Personally, I find this sort of thing immensely helpful.

- A *paned window* is a window divided into vertical or horizontal panes, which can be resized by the user. A well-known example of this is Netscape's mail interface, or Adobe's acrobat reader, where you can change the size of the pane on the left (the mailboxes or the bookmarks) by grabbing the dividing line and moving it horizontally. For its usage, see `PanedWindow` and the example `MainPanedWindow.hs` in `htk/examples/tix`.

- A note book has a number of pages with tabs on top. It displays one page at a time, selected by the tab. The pages are represented by the type `NoteBookPage`, which is an instance of the `Container` class, so you can pack widgets into them.

- A `LabelFrame` is a frame with a label; useful for grouping information together.
- A combo box is a combination of an entry and a list box, and as such has a value. Used much like list boxes, but with not quite as versatile selection schemes.

Fig. 6 shows notebooks, labelled frames and a combo box.

There are many more Tix widgets, but we have found not all of them as immediately useful as the ones above. Feel free to encapsulate more (it should not be too hard). However, if you want to use Tix' tree widgets, consider using the tree list from the toolkit (see below), which is much nicer.

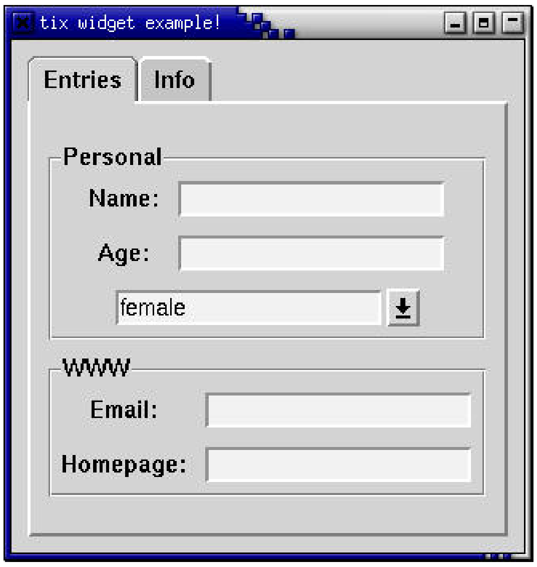


Figure 6: Tix Widgets

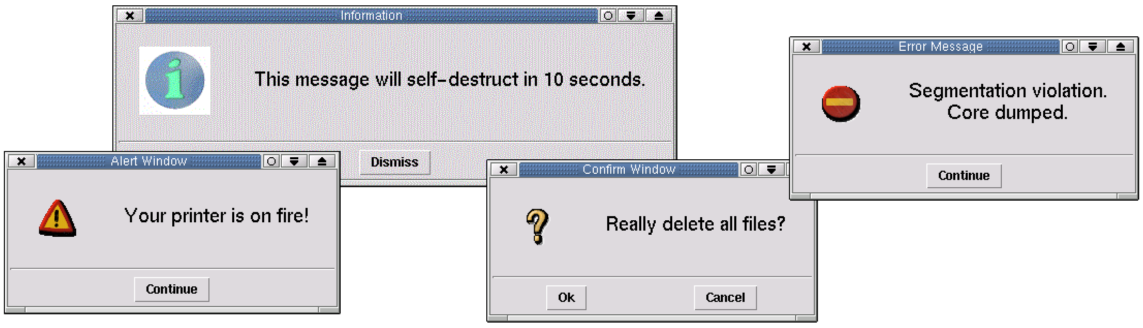
10 The Toolkit

The toolkit contains a number of so-called *mega-widgets* and *dialog windows*. Mega-widgets are are widgets which are implemented in Haskell rather than Tcl/Tk, and a dialog is a window querying the user for information. We just mention these here, without explaining them in detail, and show some screen shots. For most of them, their usage is pretty obvious.

The examples mentioned below can be found in `htk/examples/toolkit`.

10.1 Dialog Windows

The module `ModalDialog` builds modal dialogue (see Sect. 8.2) with toplevels (see Sect. 8.2). The module `DialogWin` contains an abstraction for dialogue windows, and built on top of that, frequently used windows, such as alert, information, confirmation windows and windows, and much more (Fig. 7). See the example in `Maindialog.hs`.



This message will self-destruct in 10 seconds.

Your printer is on fire!

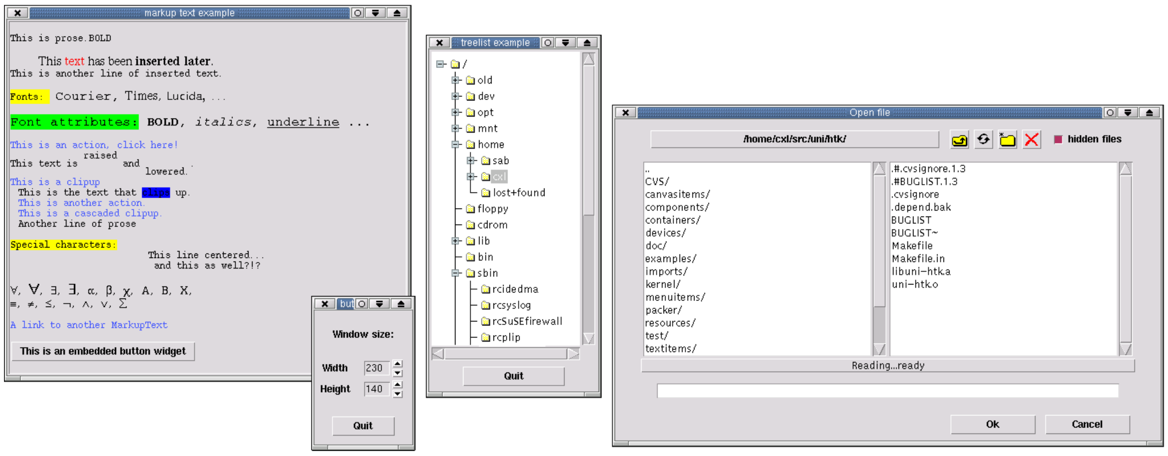
Really delete all files?

Segmentation violation.
Core dumped.

Figure 7: Dialog windows: alert, information, confirmation and error

The module `FileDialog` implements a dialog query for a file name while browsing the directory. The file dialog is vaguely in the style of GTK+, with directories on the left and files on the right. See the example in `Mainfiledialog.hs`, and the screenshot in Fig. 8.

10.2 Mega-Widgets



```
This is prose.BOLD

    This text has been inserted later.
This is another line of inserted text.

Fonts: Courier, Times, Lucida, ...

Font attributes: BOLD, italics, underline ...

This is an action, click here!
               raised
This text is          and
                          lowered.
This is a clipup
 This is the text that clips up.
 This is another action.
 This is a cascaded clipup.
 Another line of prose

Special characters:
                        This line centered...
                         and this as well?!?

∀, ∀, ∃, ∃, α, β, χ, A, B, X,
≡, ≠, ≤, ¬, ∧, ∨, ∑

A link to another MarkupText
```

```
Width  230
Height 140
```

```
/
  old
  dev
  opt
  mnt
  home
    sab
    cxl
    lost+found
  floppy
  cdrom
  lib
  bin
  sbin
    rcidedma
    rcsyslog
    rcSuSEfirewall
    rcplip
```

```
/home/cxl/src/uni/htk/

..                   .#.cvsignore.1.3
CVS/                 .#BUGLIST.1.3
canvasitems/         .cvsignore
components/          .depend.bak
containers/          BUGLIST
devices/             BUGLIST~
doc/                 Makefile
examples/            Makefile.in
imports/             libuni-htk.a
kernel/              uni-htk.o
menuitems/
packer/
resources/
test/
textitems/
Reading...ready
```

Figure 8: Toolkit components: Markup text, spin buttons, tree lists and a file dialog.

The module `MarkupText` provides an easier, more abstract and flexible way of filling an editor with markup text. For example, you can simply write

```
[prose "This is ", bold [prose "BOLD"],
 prose " and ", bgcolour [prose "yellow."]]
```

without needing to count the indices for the text tags manually. Hypertext, embedded widgets, mathematical symbols and much more are also supported. See also the example in `Mainmarkup.hs`.

A `SpinButton` is a entry with two arrows which can increment or decrement the entry's value. See the example in `Mainspinbutton.hs`.

A `TreeList` is a graphical representation of a tree structure. It allows subtrees to be hidden or shown. One popular use of this is to display the file system (see example `Maintreelist.hs`), but the possibilities are manyfold.

The module `LogWin` implements a log window, a scrolled toplevel in which text is displayed, to which the program can append. The user can clear the whole text, or save it to a file.

10.3 Notepads and GenGUI

The `Notepad` is a canvas which displays *items*. An item is given by the class `CItem`; it has a name and an icon. The notepad lets the user arrange the icons by dragging them around, and allows drag-and-drop between items.

The generic `GenGUI` interface allows the visualisation of items which are arranged in a hierarchical structure. It uses a tree list to display the hierarchy, a notepad to display all items, and an editor to display a single item selectable by double-clicking. See the examples in `htk/examples/gengui`.

10.4 Forms and Menus

The module `SimpleForm` implements forms in a simple, but typed manner. Roughly spoken, a form consists of pairs of a label and an entry, radio button, menu or other widget. The possible values entered by the user can be sanity-checked with so-called guards. See the example in `Mainsimpleform.hs`.

The module `MenuType` gives an abstract (and rather more user-friendly way) of describing menus. It does not support radio buttons yet, but potentially you might want to use to build menus rather than Tk's slightly convoluted way (remember Sect. 5). `HtkMenu` implements these for `HTk` (i.e. `MenuType` are the types, and `HTk` menu the implementation); see the example in `Mainsimplemenu.hs`.

10.5 Small Mega-Widgets

These are small but rather useful mega-widgets. We just mention them briefly:

- A `ScrollBox` is a scrollable widget inside a box scroll bars attached.

- A `Separator` is a frame with relief and borderwidth, to optically separate widgets.
- A `SelectBox` is a container for a group of buttons, one of which can be nominated as default button.
- An `IconBar` is a bar containing a row of buttons (ideally of the same size showing an image, but this is not enforced.)

References

[1] Kevin Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[3] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambrige University Press, 1999.

[4] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[5] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[6] George Russel. Events in haskell and how to implement them. In *International Conference on Functional Programming*, 2000.

[7] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1997.

A The HsMines Game

Section contributed by Christoph Grimmer `<crimson@informatik.uni-bremen.de>`

As an example for more complex GUI programming, we will now develop a GUI for a Minesweeper like game called hsMines. Just in case you do not know Minesweeper or any of its many clones I'll give a short overview.

A.1 What's that game?

Playing Minesweeper you have a grid of about 15 by 15 similar fields. Hidden inside these fields could be a mine or just an empty field – you wont know until you click at the field and are eventually shred to pieces by some nasty mine. The goal of the game is to find all the mines and mark them with tiny flags — a nearly impossible task. To make the game any fun you are told the exact amount of mines around the field you just opened. If this number is 0, it's safe to explore all the adjacent fields. And because this is a stupid task, the machine does it for you. If you manage to explore all empty fields you win, if you find one of the mines, you lose, best time gets the highscore. Simple as that.

This is not a course in Haskell programming I will not spent many words on the games code itself and will come straight to the very heart of this section, the hsMines GUI. You can find the source for the game in `htk/examples/intro/MainhsMines.hs`, in case you are curious.

A.2 In the beginning is the Window

As we have read, it all starts with a window. And since we are running compiled code here, it's created in the `main` function.

```
main =
 do htk<- initHTk [withdrawMainWin]
    run htk normalSize

run :: HTk-> (Int, Int)-> IO ()
run htk currentSize =
 do main <- createToplevel [text "hsMines"]
```

Having only this, the GUI would look rather boring, but it's not bad for just a few lines of code, isn't it? It looks a bit awkward but we will see later that this is needed to make the field resizeable. If this were all the program, we would need a line

```
    finishHTk
```

to clean up everything and finish. With that, our program would like shown in Fig. 9.

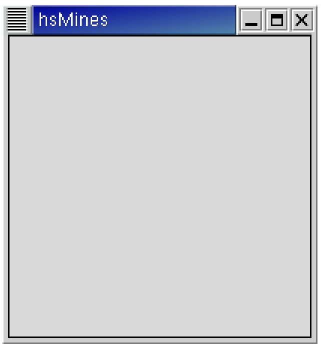


Figure 9: The hsMines window, initial state.

A.3 Menus

Let's start start building the interface, then. Every interface needs menus, so let's create some.

```
    menubar <- createMenu main False []
    main # menu menubar

    fm <- createPulldownMenu menubar [text "File"]

    restb <- createMenuCommand fm [text "Restart"]
    quitb <- createMenuCommand fm [text "Quit"]

    pm <- createPulldownMenu menubar [text "Preferences"]

    pmc1 <- createMenuCascade pm [text "Size"]
    pmc1m <- createMenu main False []
    pmc1 # menu pmc1m

    pmc2 <- createMenuCascade pm [text "Difficulty"]
    pmc2m <- createMenu main False []
    pmc2 # menu pmc2m
```

Let's go through this step by step. In line 1 we create a menu inside *main* (our window) that was created in `run` above. In line 2 we tell HTk to attach this new `menu` we call *menubar* to be the `menu` of *main*.

In line 3 (we will not count the empty lines), we create our first pulldown menu — that is what actually is normaly called a menu. This pulldown menu is created inside *menubar*, is called *fm* and has the charming text 'File'. To fill this menu empty, we create menu entries, called menu commands inside *fm*; a menu command is the simplest form of menu item which you can just select (or not). So by now we have *restb* and *quitb* inside *fm* inside *menubar* inside *main*.

Besides restarting and quitting, we will need to put some more functionality into our menu. As we read above, the game grid will be resizeable so we will need a 'Preferences' menu to have these commands in.

We create a second pulldown menu in *menubar* called *pm*. In this menu we nest to submenus (called menu cascades). Each of the cascades has a name, *pmc1* and *pmc2*, and a text configuration set to its title. The next step is a bit tricky. One would now expect to fill the cascades directly with some commands. But the `menu cascades` are only `Containers` holding other menus, so we have to create another two menus inside *main* and assign them to the two cascades. These two menus are called *pmc1m* and *pmc2m* which should be an abreviation for preference menu cascade first menu (and second respectively).

By now we have a menu bar which holds two pulldown menus. The first contains two commands, the second contains two cascades which in turn each contain a menu again. To make sense of these two submenus we have to fill them of course. And finally we have to put some functions behind all those commands or this would all be for naught.

```
varSize <- createTkVariable currentSize
sr1 <- createMenuRadioButton pmc1m
        [text "tiny (6x6)", value tinySize,
         variable varSize]
sr2 <- createMenuRadioButton pmc1m
        [text "small (10x10)", value weeSize,
         variable varSize]
sr3 <- createMenuRadioButton pmc1m
        [text "normal (15x15)", value normalSize,
         variable varSize]
sr4 <- createMenuRadioButton pmc1m
        [text "large (20x20)", value bigSize,
         variable varSize]
sr5 <- createMenuRadioButton pmc1m
        [text "huge (25x25)", value hugeSize,
         variable varSize]

varDiff <- createTkVariable (6:: Int)
dr1 <- createMenuRadioButton pmc2m
```

```
        [text "easy", value (8::Int),
         variable varDiff]
dr2 <- createMenuRadioButton pmc2m
        [text "normal", value (6::Int),
         variable varDiff]
dr3 <- createMenuRadioButton pmc2m
        [text "hard", value (4::Int),
         variable varDiff]
dr4 <- createMenuRadioButton pmc2m
        [text "nuts", value (3::Int),
         variable varDiff]
```

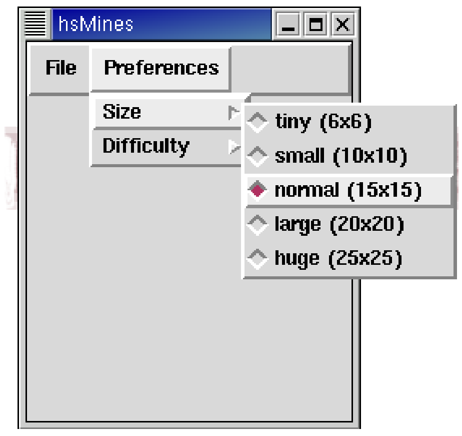


Figure 10: The hsMines main window with an open menu

In the code above we do several new things. First we create two Tk variables called *varSize* and *varDiff*. These are necessary to hold the state of the GUI, in this case the user's selection. Our next step is to create a menu radio button in each of the two submenus. The first radio button assigns pairs of integers to the TkVariable *varSize*, the actual values are as indicated in the text configuration and given in a couple of functions above in the code which can be fully seen in the source. Fig. 10 shows the size submenu of the preferences.

As you we can see in the code above (in `main` and `run`), *currentSize* holds the value of *normalSize* and so by default the radio button is set to *normalSize*. Nearly the same happens with *varDiff* and its radio button.

Note that we have to resolve the overloading of the numeric constants by type annotations, because TkVariables can hold all instances of the class `GUIValue`; when we just write `value 3`, this might also be e.g. the floating point number 3.

But something should happen we click those menus, so we need to bind them to some events:

```
restartClick <- clicked restb
quitClick <- clicked quitb

csr1 <- clicked sr1
csr2 <- clicked sr2
csr3 <- clicked sr3
```

```
    csr4 <- clicked sr4
    csr5 <- clicked sr5
```

This binds the commands and the size radio button(s) to a couple of `events`, which will occur whenever one of the buttons is selected (i.e. clicked). Note that we do not bind anything to the difficulty submenu. This is because we are actually not interested in the user selecting a new difficulty during the game, we just read out the value set by the user every time we restart a game. (This is not very polite — it might be better to inform the user of this when he changes the selection, but this is just a small demonstration program.)

A.4 The field

To make all the decoration perfect, we need a little smiley atop the playfield which can be used to restart the game.

```
    sm <- newButton main [photo smSmileImg]
    startClick <- clicked sm

    pack sm [Side AtTop, PadY 20, PadX 20]
```

We create a `Button` called *sm* (from **sm**iley btw), bind it to an event, and pack it at the top of the `main` window, padding it 20 pixels wide in both directions. The `photo` configuration assigns an image to any widget that can hold one, such as a button. In this case we have a collection of base64 encoded GIFs directly in the code (not shown here); [7] this has the advantages of making the code stand-alone. As we will see later, there are also tiny `Images` for the empty field and the numbers and not only the flag; the main reason for this is to that all buttons should always have precisely the same size. We further have a wide selection of colours for the numbers. Just try to find out wich colour the 8 has. . . But back to the smiley.

Same as the `menu commands`, the `Button` is useless by itself. To get things started we create a channel named *restartCh*. We will set things up such that sending something (anything, really) over the channel will restart the game.

```
    restartCh <- newChannel

    bfr <- newFrame main [width (cm 10)]
```

[7]The images have been taken from gnomines (the GNOME minesweeper clone), where they are attributed to `tigert` (Tuomas Kuosmanen).

But before we can initialise the game field, we of course have to create it. To contain the field we create a new a frame with a given width. The width is just to have something to start with and will be adjusted by the packer as needed. The frame is packed below the smiley button. They both are told to be at the top of *main*, but because it is packed after the button it is placed below.

```
    pack bfr [Side AtTop, PadX 15]
```

Until now this was all very plain and straight. The buttons for the field are created in a more complex way. The function `buttons` is used to create a number of buttons along with their position, and assigns them to *allbuttons* (i.e. *allbuttons* has type `[((Int, Int), Button)]`. To pack them, we have to iterate through the whole list using `mapM_`. And in order to have all buttons appear on the screen at once, and not one by one, we wrap `delayWish` around this.

```
    size <- readTkVariable varSize
    allbuttons <- buttons bfr sm (receive restartCh) size
    delayWish $ mapM_ (\(xy, b)-> grid b
          [GridPos xy, GridPadX 1, GridPadY 1]) allbuttons
```

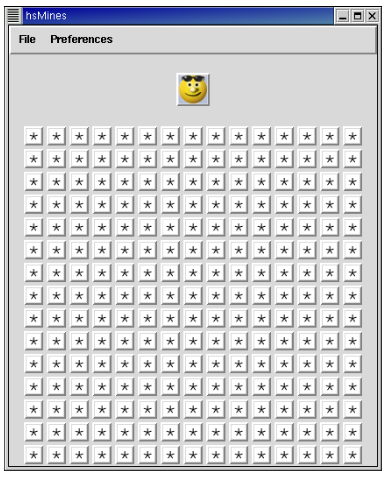


Figure 11: The uninitialised hsMines field

In this case the grid packer is used to put all the buttons at exactly stated positions in a grid. If the code compilation does not reach any point where the game is started, the packed but uninitialised game field including the smiley and all thing looks like this. To make a distinction between uninitialised and initialised fields, we put a little star on each `Button`. The result is shown in Fig. 11.

All that is left to do now in `run` is to initiate the game for the first time. The actual playing happens in the event handling of the buttons.

```
    let start :: IO ()
        start = do diff <- readTkVariable varDiff
```

```
                 sendIO restartCh diff

  -- start the menu handler
  stopmh<- spawnEvent (forever
        (startClick >>> start
      +> quitClick >>> destroy htk
      +> choose [csr1, csr2, csr3, csr4, csr5] >>>
           createMessageWin "Changes come into effect
             after \"Restart\"." []))

  -- the restart handler (note no forever!)
  spawnEvent (restartClick >>>
                do stopmh
                   destroy main
                   nuSize <- readTkVariable varSize
                   run htk nuSize)

  -- start the game
  start

  -- wait for game to stop, then clear up the mess
  finishHTk
```

There is a bit more to clean up the window and such; let us go through this step by step. `start` is a function that reads the variable *varDiff* and sends the value over the channel `restatCh`. This has the effect of restarting the game (see `buttons` below, the event of receiving something on that channel restarts the game).

First, we build a composed event that handles the menu buttons: clicking the smiley (event `startClick`) should restart the game, and selecting the quit menu (event `quitClick`) should finish the game by destroying the main window. Selecting a new size (`csr1` to `csr5`) should not do anything immediately, but we are polity and inform users that they have to restart the game— clicking just the smiley is not enough!)

On the other hand, a separate event handles the restart. This is because if we restart we destroy the main window, and start the game again from the top. From the interface design point, this is bad design for two reasons: firstly, it makes the interface very jittery with windows unecessary opening and closing and what not, and secondly, there is the awkward handling of having to restart the game with the menu if you want to change the size.

A.5 The `buttons` function

This function has a rather complex signature.

```
buttons :: Container par=> par-> Button-> Event Int
                           -> (Int, Int)
                           -> IO [((Int, Int), Button)]
```

For reference recall how it was called:

```
    allbuttons <- buttons bfr sm (receive restartCh) size
```

We have a class constraint on the first argument, *par*, which has to be container. *par* happens to be just that, a `Frame`. Lucky us. The next argument has to be a button as we can happily admit our smiley button *sm* is. The third argument has to be an event, more precisely `Event Int`. This event signals that the game should be restarted with the difficulty level given; in the call, this is the event which signals receiving something in the *restartCh* channel (recall from above how we send something along this channel to restart the game).

`buttons` creates all the buttons, and returns a list of pairs, where each of these pairs contains the coordinates (as a pair of integers) and the button. We can see that it is simpler to examine the signature ourself than try to puzzle out what we just read.

The code of `buttons` is simple at start.

A.5.1 Creating an array of buttons

```
buttons par sb startEv (size@(xmax, ymax)) =
  do buttons <- mapM (\xy->
       do b<- newButton par [photo starImg, relief Raised]
          return (xy, b)) [(x, y) | x <- [1.. xmax],
                                    y <- [1.. ymax]]
```

This code creates all the `Buttons` and paints a little star inside so the playfield looks as shown in Fig. 11, using standard list comprehension and map for monads. But there is more to happen in the `buttons` function!

A.5.2 Binding the buttons

```
     let bArr = array ((1,1), size) buttons
         getButtonRelease b n xy =
            do (click, _) <- bindSimple b
                              (ButtonRelease (Just n))
               return (click >> return xy)
     leCl <- mapM (\(xy, b)-> getButtonRelease b 1 xy)
                                 buttons
     riCl <- mapM (\(xy, b)-> getButtonRelease b 3 xy)
                                 buttons
     press <- mapM (\(_, b)->
       do (cl, _)<- bindSimple b (ButtonPress Nothing)
          return cl) buttons
```

This looks rather complicated but does nothing more than what we did above when we bound the smiley button to the event *startClick*. First, we arrange our buttons in an array so we can later on refer to the button at position (x, y) easier. Then we bind three events to each of the buttons: one for releasing the first (left) button, one for releasing the third (right) button, one for pressing any button. The left and right clicks are of type `Event (Int, Int)`, because we will later on have refer to the coordinates of a button being released.

A.5.3 Starting the game

We now define three events (and a couple of auxiliary functions) which encode the main logic of the game. The first one starts the game, the third and second play the game. Then, if we synchronise on the start event, we set the game in motion and in effect wait until it is over.

All of these definitions are local to `buttons` and use the declarations from above, such as *leCl*, *riCl* and *bArr*.

```
let start :: Event ()
    start =
      startEv >>>= \d->
        do m <- createMines (snd (bounds bArr)) d
           sb # photo smSmileImg
           mapM_ (\b-> b # photo zeroImg >>=
                           relief Raised) (elems bArr)
           sync (play m)
```

This says: after the start event (the argument of which is d here, the difficulty level), we create the mines on the playing field, make the smiley smile, and fill all buttons with the zero image. Have a look at how we create the mines later, it is of no importance for the GUI. Then we play.

A.5.4 Playing the game

```
play :: Mines-> Event ()
play m
  = do r <- choose leCl >>>= open bArr m
       case r of Nothing -> always gameLost
                   >> gameOver
                 Just nu -> playOn nu
    +>
    do r<- choose riCl >>>= flag bArr m
       playOn r
    +>
    do choose press
       always (sb # photo smWorriedImg >> done)
       play m
    +>
    start
```

Playing is easy. The `Mines` datatype needs not to be explained right now, suffice it to say that it models the state of the playing field (including the mines, but also keeping track of which fields have been explored or flagged). To play with a set *m* of mines means to execute these steps over and over again:

- If the left mousebutton is released, call `open` with the button array, the mines and the position of the field we want to open (note clever η-reduced notation). If `open` returns `Nothing`, we lose; otherwise, we play on with the new playing field.
- If the right mousebutton is released, call `flag` with the button array, the mines and the position. No evil may occur, just play on.
- If a mouse button is pressed (remember, even the middle button counts), the smiley should look worried. Normally any of the two release events above will occur within a short while so the smiley changes back, but if we press the middle button, the smiley stays worried. We consider this a feature, and in the best academic tradition leave it to the reader to come up with a solution.
- On the other, if a start event occurs, restart the game; this can happen e.g. if the users clicks the smiley in the middle of the game.

```
playOn :: Mines-> Event ()
playOn m = do always (sb # photo smCoolImg)
              if all (not.untouched) (elems m) then
```

```
                                          do always gameWon
                                             gameOver
                              else play m
```

playOn takes care of the smiley when we (de)flagged or explored a field. It also checks wether we have won with the previous move; this is the case if there are no untouched fields left (i.e. all fields are either cleared or flagged). This works because we are only allowed to drop as many flags as there are mines on the field.

A.5.5 Game Over

gameLost and gameWon just open message windows to tell you that you've lost or won. The smiley also takes appropriate action, feeling very sick or grinning inanely.

```
gameLost :: IO ()
gameLost =
  do sb # photo smSadImg
     createAlertWin "*** BOOM!***\nYou lost." []
gameWon :: IO ()
gameWon =
  do sb # photo smWinImg
     createMessageWin "You have won!" []
```

The windows are modal, so only after you closed them, the game is really over and can be started again.

```
gameOver :: Event ()
gameOver = start
           +> (choose (leCl++ riCl) >> gameOver)
           +> (choose press >> gameOver)
```

The play event changes into gameOver once the game is over. It only lets you restart the game, and just swallows any of the button events (leCl etc). If we didn't react to the button events like that, they would still be in the event queue, and be reacted to once the game restarts— not what you want really!

Anyway, now we have set up the logic to play the game we can spawn an event handler which waits for the game to start by synchronising on the start event. All that is left is to return the buttons so the main function can pack them. (Note that we bound events to buttons before packing them, this is entirely possible.)

```
        spawnEvent start
        return buttons
```

This is the end of the buttons function. Everything else is just plain haskell. Ok, you're right, there is some tiny bits left. Nobody explained how the numbers show up when a non-mine field is explored, right? Okay, we'll come to that now.

A.6 Modelling the Playing Field

The playing field is represented by two arrays: one contains just the buttons, since these are not going to change, and one contains the actual state of a field, which is going to change:

```
data State = Cleared Int
           | Unexplored { flagged :: Bool,
                          mine    :: Bool }

type Mines   = Array (Int, Int) State
type Buttons = Array (Int, Int) Button
```

Three utility functions, untouched, mines and flags, check wether is a state is untouched (neither cleared nor flagged), and count the number of mines or flags on a field; we do not show them here.

Now, the simple part is leaving and taking flags.

```
flag :: Buttons-> Mines-> (Int, Int)-> IO Mines
flag b m xy =
  case m!xy of
    Cleared _ -> return m
    s@(Unexplored{flagged= f})->
        if f || (sum (map flags (elems m)) <
                    sum (map mines (elems m)))
        then do b!xy # (if not f then photo flagImg
                        else photo zeroImg)
                return (m // [(xy, s{flagged= not f})])
        else return m
```

If the the field is *Cleared* which means it was explored earlier the mines are left unchanged. Mines are simply returned. If the field is Unexplored and flagged we set it unflagged and assign the *zeroImg* to its Button and vice versa. However, when dropping a flag we have make sure that there are not more flags on the field than mines.

The more complex part is actually exploring the field.

```
open :: Buttons-> Mines-> (Int, Int)-> IO (Maybe Mines)
open b m xy =
  case m!xy of
    Cleared _                -> return (Just m)
    Unexplored {flagged= True} -> return (Just m)
    Unexplored {mine= True}    -> return Nothing
    _ -> peek b m [xy] >>= return. Just
```

The function takes the same arguments but it returns a `Maybe` of mines. If we try to explore a `Cleared` field, nothing changes. If we try to explore an `Unexplored` field which is flagged, we return the given argument and still nothing changes, because it is inconvenient to accidently click a flag and get killed. If we try to open an `Unexplored` field which is a mine the function returns `Nothing`, and we have lost. And at last, when every other case is weeded out, we `peek` inside an `Unexplored`, nonflagged nomine field.

```
peek :: Buttons-> Mines-> [(Int, Int)]-> IO Mines
peek b m [] = return m
peek b m (xy:rest) =
   let adjMines :: Int
       adjMines = sum (map (mines. (m !)) (adjacents m xy))
       nu       = m // [(xy, Cleared adjMines)]
   in do (b!xy)# photo (getImg adjMines) >>= relief Flat
         if adjMines == 0 then
            peek b nu (rest ‘union‘
                      (filter (untouched. (m !))
                              (adjacents m xy)))
            else peek b nu rest
```

`peek` takes a list of coordinates to check. Initially this should be exactly one coordinate, namely the one we try to explore right now. `adjMines` calculates the number of mines around the field we explore; `adjacents` is a utility function which takes a positition and returns the list of its adjacent positions.

We assign the corresponding image to the button’s photo and change its relief to flat (note how we compose configurations with `(>>=)`). If the number of mines in the adjacent fields is 0 we can savely explore all of these. So *rest*, the list of coordinates still to explore is united with the coordinates of the adjecent fields which are still untouched. If we forgot to apply this filter the exploration would go on forever exploring the same fields over and over again, but this way the program explores all

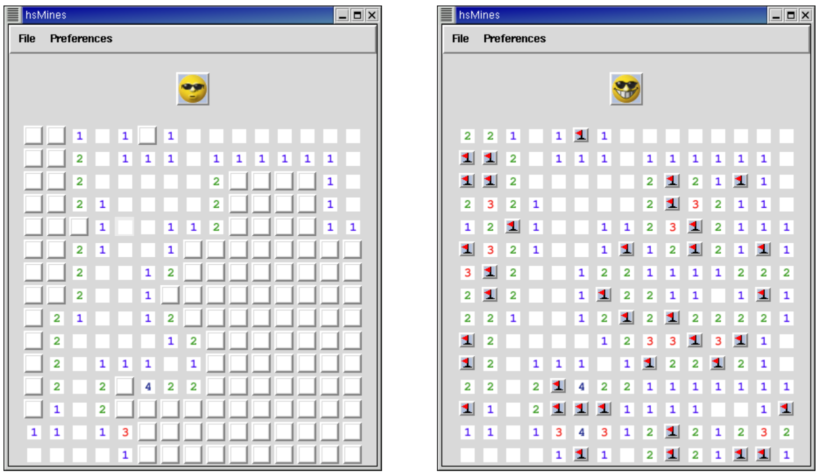


Figure 12: hsMines after one lucky click, and a few lucky clicks later.

safe fields, creating empty areas surrouned by numbers on the playfield (see Fig. 12, left).

And just in case you don't believe this for real I finished the game ;-) (Fig. 12, right).

The excurse through the `hsMines` source is over for now. There is still a couple of interessting functions (such as `createMines`) left to explore. Have fun.